Formal Verification of Systems Software

No Execution of Malicious Software in Linux in Networked Embedded Systems

Formell verifiering av systemmjukvara

Ingen skadlig mjukvara i Linux i uppkopplade inbyggda system

Jonas Haglund

E-mail: jhagl@kth.se

Examensarbetets ämne: Datalogi i masterprogrammet datalogi

Handledare: Roberto Guanciale

Examinator: Johan Håstad

Datum: 2016-11-05

Abstract

Embedded systems provide critical services in numerous applications requiring complex functionality. The complex functionality is often implemented by complex software stacks, such as GNU/Linux. Since complex software often contains bugs, some of which might be exploitable by attackers, embedded systems are exposed to security attacks. For instance, malicious software might be injected by exploiting a buffer overflow bug. In addition, the number of embedded systems connected to the Internet is increasing, enabling attackers to perform their attacks remotely and thus making embedded systems even more exposed to security attacks.

To address this issue, this thesis presents a software design and an implementation that host Linux with Internet support, such that only signed (non-malicious) Linux code is executed. The software design consists of three software components: A hypervisor and two guests, Linux and a monitor. These three software components can be executed on the development board BeagleBone Black, containing an ARMv7 CPU and a network interface controller (NIC). The hypervisor ensures that the three software components are securely separated and the monitor ensures that only signed Linux code is executed. The software design and the implementation take into account that Linux code might instruct the CPU to configure the NIC to access memory. This software design and implementation therefore ensure that the security service provided by the monitor is not breached by the NIC. In order to increase the reliability of this system, a pen-and-paper proof plan is presented with the purpose of guiding a formal proof of that only signed Linux code is executed in this system.

The original software design and implementation of the hypervisor, provided by the PROSPER project, did not have NIC support. The software design and the implementation of the hypervisor therefore have been extended with a security layer that intercepts writes to NIC registers that are performed by the CPU when it is executing Linux. If a NIC register write cannot enable unsigned (malicious) Linux code to be executed, the hypervisor lets the NIC register write take effect, and otherwise blocks it. In addition, the original software design of the monitor, also provided by the PROSPER project, did not consider the operation of the NIC. The design of the monitor therefore has also been extended in order to prevent memory configurations of Linux that enable unsigned Linux code to be executed. The proof plan describes and motivates how it can be formally proved in a theorem prover that only signed Linux code is executed in this system. For the purposes of the proof plan, HOL4 models of the hardware have been identified and a formal model of the NIC has been specified in pseudocode.

If the work presented in this thesis is fully implemented and combined with earlier work from the PROSPER project, a networked embedded system is acquired in which, with high reliability, no malicious Linux code is executed.

Sammanfattning

Inbyggda system har ofta kritiska roller som kräver komplex funktionalitet. Denna funktionalitet implementeras ofta genom återanvändning av komplexa mjukvarustackar som exempelvis GNU/Linux. Då antalet buggar ökar med mängden kod och dess komplexitet, så finns det risk för att inbyggda system innehåller buggar, där vissa buggar potentiellt öppnar upp säkerhetshål. Exempelvis kan en buffertöverskridningsbugg möjliggöra installation av skadlig mjukvara. Därtill är det vanligt att inbyggda system är anslutna till Internet vilket gör inbyggda system än känsligare för säkerhetsattacker.

Denna uppsats beskriver en mjukvarudesign och dess implementation som ökar inbyggda systems tillförlitlighet. Implementationens syfte är att försäkra att endast signerad (icke-skadlig) Linuxkod exekveras. Mjukvarudesignen består av tre mjukvarukomponenter: En hypervisor och dess två gäster, Linux och ett kontrollprogram. Dessa tre komponenter kan exekveras på utvecklingskortet BeagleBone Black som har en ARMv7 processor och ett nätverkskort. Hypervisorn försäkrar att de tre mjukvarukomponenterna är isolerade från varandra på ett säkert sätt, och kontrollprogrammet försäkrar att endast signerad Linuxkod exekveras. Mjukvarudesignen och implementationen förhindrar processorn när den exekverar Linux från att konfigurera vilka minnesåtkomster nätverkskortet kan göra. Nätverkskortet kan därför inte förhindra kontrollprogrammet från att försäkra att endast signerad Linuxkod exekveras. För att öka detta systems tillförlitlighet så presenteras även en bevisplan som beskriver hur ett formellt bevis kan konstrueras för att endast signerad Linuxkod exekveras i detta system.

Den ursprungliga mjukvarudesignen och implementationen av hypervisorn, utvecklade i PROSPER projektet, hade inget stöd för nätverkskortet. Mjukvarudesignen och implementation av hypervisorn har därför utökats med ett mjukvarulager som anropas när processorn exekverar Linux och försöker skriva ett nätverkskortsregister. Om skrivningen till nätverkskortsregistret inte kan möjliggöra exekvering av osignerad (skadlig) Linuxkod, så utför processorn skrivningen, och annars inte. Även den ursprungliga mjukvarudesignen av monitorn, också utvecklad i PROSPER projektet, tog inte hänsyn till nätverkskortet. Mjukvarudesignen av monitorn har därför också utökats för att försäkra att minneskonfigurationen är kompatibel med nätverkskortets konfiguration så att nätverkskortet inte möjliggör exekvering av osignerad Linuxkod. Bevisplanen beskriver och motiverar hur det kan bevisas formellt i en teorembevisare att endast signerad Linuxkod exekveras i detta system. Denna bevisplan baseras på en mängd HOL4 modeller och en formell modell av nätverkskortet som specificerats med pseudokod.

Om arbetet som presenteras i denna uppsats implementeras fullt ut och kombineras med tidigare arbete som gjorts i PROSPER projektet, så erhålles ett uppkopplat inbyggt system som, med hög tillförlitlighet, endast exekverar signerad Linuxkod.

Table of Contents

1 Introduction

Embedded systems are widely adopted in today's society and encountered in many different sorts of devices and systems. Examples are smart wristwatches, surveillance systems, networking equipment, smartphones, tablets, smart TVs, fridges, medical devices, cars and financial transaction devices, but also more critical systems such as air and railway traffic control systems, nuclear power plants and defense systems.

The role of embedded systems and their ubiquity make them both critical and heavily exposed to attacks. Considering the kind of vulnerabilities that exist or have existed, and exploits that have occurred, security in embedded systems must be taken seriously as exemplified by the following articles and reports:

- A study [4] was made of the IT security of several hospitals in USA in 2012. It was found that drug infusion pumps and defibrillators could be remotely accessed and cause danger to patients, and that digital medical records could be accessed and manipulated. Even though not all of these devices were directly connected to the Internet, many of them were connected to internal networks that in turn were connected to the Internet.
- It has been reported [5] that there are significant numbers of vulnerabilities in industrial control systems that control critical infrastructures. Several attacks of such systems have also been successful. For instance, customers of these systems have downloaded malicious software [6], a blast furnace in a steel plant was set in a dangerous state causing damage to the factory [7], and in a dam near New York hackers took control of the flood gates [8].

  Several incidents have occurred in nuclear power plants [9]. One attack by a malicious program prevented personnel from viewing data of temperature sensors and radiation detectors. This attack in combination with other weaknesses found in another nuclear power plant could result in disrupted system operation and in the disabling of sensors that indicate problems to personnel. Some attacks were also successful on systems that were physically isolated from the Internet.
- Modern cars are hackable advanced computer networks, where hackers can take control of cars to change the speedometer, activating and disabling the breaks, or turning the steering wheel [10, 11].
- An investigation of air traffic control security concluded that the Internet connectivity of modern airplanes can lead to unauthorized access to aircraft software [12]. The reason is that the cockpit software is only isolated from the rest of the entertainment systems and the Internet by means of a firewall. If a hacker successfully compromises the firewall, the hacker can access the network that the cockpit software is connected to.
- Examples of malicious programs for smartphones can install new programs, delete files, and transmit personal data, including banking information, to remote servers [13, 14].

In addition, the widespread use of embedded systems will probably only increase in the future. If hardware continues to evolve with additional CPUs and larger memory capacity, a single embedded system can be used to execute more applications concurrently and new applications that require higher performance. Also, if the Internet of Things continues to gain success, the increased connectivity of embedded systems will increase and probably result in that embedded systems will be used in new environments. Cisco, Ericsson and Huawei predict that there will be more than 25 billion connected devices by 2020 [1-3]. The development of hardware and the Internet of Things will therefore probably make it more common that critical and non-critical applications will run on the same chip or on connected chips. Hence, the attack surface of critical applications increases since the hardware they execute on can be affected by other applications, which might contain malicious code or bugs. For instance, in a car the software that handles the ABS brakes and the airbags might run on a chip that is connected to the Internet or run on the same chip as the entertainment software. The safety software might therefore be affected by potential bugs and malicious code injected in the entertainment software. Hence, the security for embedded systems must be taken even more seriously in the future.

In many embedded systems Linux is used as the operating system. For instance, Linux runs in all of the kinds of devices mentioned in the opening paragraph of this chapter [15-27], and according to the Linux Foundation, Linux is the most widely used software in the world [28]. Some reasons for the widespread use of Linux in embedded systems are: its small memory usage; its support for CPU architectures and I/O devices; its support for multithreading and multiprocessors; its support for common network protocols and file systems; it is customizable to include only the features needed such as device drivers, networking protocols and file systems; its support for graphics and off-the-shelf applications that are often needed in embedded systems; it is free and open source; and many developers are familiar with Linux [88, 89]. Because of these properties of Linux, it is probable that Linux will be continued to be used in embedded systems in the future. However, one weakness of Linux is its large code size which probably means that Linux contains a number of bugs, some of which might be possible to exploit by hackers.

Embedded systems are also commonly implemented by means of ARM CPUs. ARM CPUs are commonly used in embedded systems because of their low power consumption, performance and price [90, 91]. In 2015, 15 billion chips with ARM processors were sold, giving ARM a market share of 32%. About 45% of these chips were in mobile devices, and the rest in other kinds of products such as networking infrastructure and safety systems in cars [29]. ARM CPUs are also designed for other applications, such as medical and industrial systems [30].

Considering (i) the current and predicted widespread use of embedded systems, (ii) their connected and critical use, (iii) the kinds of successful attacks placed on them and the kinds of threats they are exposed to, and (iv) their common implementation with Linux and ARM CPUs, it is relevant to improve the security in networked embedded systems that run Linux on ARM CPUs. Since a large part of the modern society uses embedded systems of this kind, such an improvement is of interest to a large number of people.

1.1 Prevention of Malicious Software in Linux on ARM Processors

The PROSPER project attempts to improve security in embedded systems by developing a software platform for embedded systems and formally verifying security related properties of this software platform. Current efforts of PROSPER are devoted to preventing execution of malicious software in an embedded Linux system with and ARM CPU and Internet access, and formally verifying this execution property. The use of formal verification gives a high trustworthiness of that no malicious software is executed. This execution property and the trustworthiness of its enforcement combined with the capability of Linux to run commonly used applications and being connected to the Internet, makes such a system both dynamic and reliable. Such a system is therefore attractive for many critical applications. So far PROSPER has developed, implemented, and formally verified a software design that ensures that only signed Linux code is executed in a system consisting of one ARM CPU with memory without Internet access [86]. This work provides the foundation for this thesis and is introduced in this section, and is further described in Subsection 2.3.4.

The implementation of the software design targets the ARMv7 instruction set architecture (ISA), and involves the following three software components:

- A hypervisor executed in privileged mode. The hypervisor enables Linux and a monitor to be executed on top of it, and ensures that the executions of these three software components cannot affect each others' state insecurely.
- A Linux kernel that is paravirtualized (modified) to enable it to be executed on top of the hypervisor in non-privileged mode. The Linux kernel and the applications running on top of it are untrusted in the sense that their code could have any malicious intention. For instance, a hacker might try to insert malicious code by means of a stack overflow attack to take control of Linux.
- A monitor executed on top of the hypervisor in non-privileged mode. The purpose of the monitor is to ensure that only signed Linux code is executed.

All three software components have statically allocated and separated memory regions. Figure 1 illustrates the structure of this implementation. The two critical mechanisms of this design and implementation of ensuring that only signed Linux code is executed are the mechanisms ensuring (i) that the three software components are securely separated, and (ii) that all executable Linux code is signed. The mechanism providing the second property depends on the mechanism providing the second property. The first property is referred to as the separation property, and the second property is referred to as the execution property. The mechanism providing the separation property is described first and then the mechanism providing the execution property.

The hypervisor configures the memory management unit (MMU) to use a certain set of page tables. Those page tables are located in the memory region allocated to Linux but are mapped as read-only in non-privileged mode. Since the hardware can only be configured in privileged mode and only the hypervisor is executed in

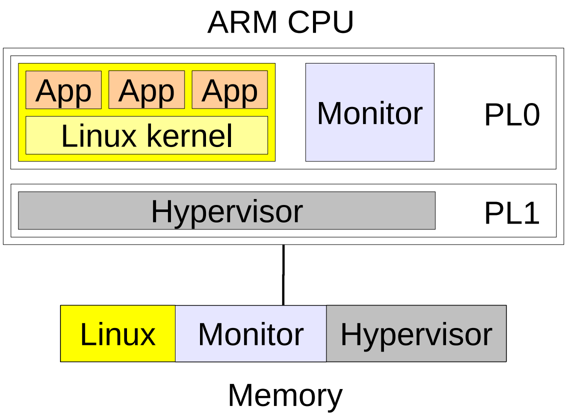


Figure 1: Linux and the monitor executed on top of the hypervisor on an ARM CPU. The hypervisor is executed in privileged mode (PL1), while the monitor and Linux are executed in non-privileged mode (PL0). All three software components have their own statically allocated memory regions.

privileged mode, the MMU and the page tables can only be configured by the hypervisor. The hypervisor configures the page tables such that Linux and the monitor can only perform the memory accesses (read, write or execute) they are supposed to perform. When Linux or the monitor attempts access a memory location, the memory management unit (MMU) traverses the page tables to determine whether the access shall be accepted or rejected. If the page tables specify that the access shall be rejected, the MMU rejects the access. Hence, the secure separation between the hypervisor, Linux and the monitor relies on the MMU and the page tables.

Since Linux creates and terminates application processes and dynamically allocates and deallocates their memory, Linux must be able to configure its virtual to physical address mapping. The software design that the hypervisor and the monitor implement therefore specifies a set of functions (hypercalls) that allow Linux to configure its memory mapping. Those functions are referred to as the memory mapping request handlers. When Linux needs to configure its memory mapping, Linux invokes a memory mapping request handler, implemented both by the hypervisor and the monitor. The handler checks that the requested configuration of the page tables does not break the separation property nor the execution property. If the requested configuration preserves both properties, the request is executed and otherwise rejected.

When a handler is invoked, the hypervisor checks if the request breaks the separation property: If the request specifies a memory mapping that gives Linux access to memory belonging to the hypervisor or the monitor, or writable access to a page table, the request is rejected. Otherwise the request is forwarded to the monitor, which checks if the request breaks the execution property: If the request specifies unsigned Linux code to be mapped as executable, the request is rejected. To ensure that all executable Linux code is signed, the monitor accepts a request if and only if the request satisfies the following two conditions:

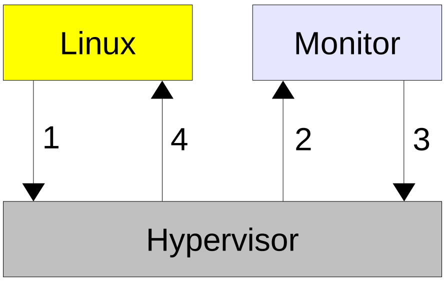


Figure 2: The interaction between the software components when Linux invokes a memory mapping request handler. The first step consists of Linux invoking a memory mapping request handler to provide a request to the hypervisor that specifies how the memory mapping shall be configured. If the specified configuration gives Linux access to hypervisor or monitor memory, or writable access to a page table, the hypervisor rejects the request. Otherwise the second step is performed where the request is forwarded to the monitor. The monitor checks that the request does not enable execution of unsigned Linux code. In the third step the monitor provides its answer. If the monitor accepts the request, the hypervisor executes it and otherwise rejects it. In the last step the hypervisor gives the CPU to Linux, which then continues its execution.

- The request does not specify a memory mapping that maps a 4 kB physical memory block (the size of the smallest page frame) as both writable and executable.
- Each 4 kB physical memory block that is requested to be mapped as executable contains signed code. The monitor determines a block to contain signed code if the digital signature of the contents of the block is in the golden image. The golden image is a set of signatures maintained by the monitor and represents the code that is considered non-malicious.

The first condition prevents Linux code from writing executable blocks, and the second condition ensures that blocks being mapped as executable initially have signed contents. The executable blocks therefore always have signed contents.

The monitor then provides its answer to the hypervisor. If the monitor accepts the request, the hypervisor executes it and otherwise rejects it. Figure 2 summarizes the interaction that occurs between the software components when Linux invokes a memory mapping request handler. Note that the mechanism of ensuring that all executable Linux code is signed also relies on the MMU and the page tables.

Since malicious code injection attacks are commonly performed by means of sophisticated techniques, it is desirable to formally verify that only signed Linux code is executed at the ISA level. Such verification takes into account: the state of the hardware, how CPU instructions modify the hardware states, and therefore the interaction between the hardware and the software. Verification at the ISA level therefore establishes that only signed Linux code is executed with extraordinary reliability, and makes the system especially suitable for security critical applications. Verification at such a detailed level is the ambition of PROSPER, but so far only the design of the memory mapping request handlers has been verified.

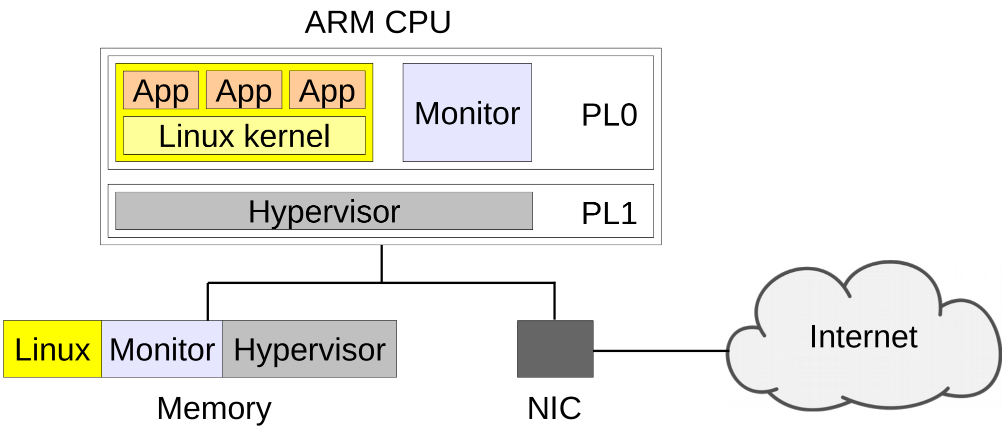


Figure 3: The original system extended with a network interface controller (NIC). This system is the focus of this thesis.

That is, how the handlers modify the hardware states, from the states from which Linux invokes the handlers to the states to which the handlers return and the execution of Linux continues. The verification of the handlers has been performed by means of the theorem prover HOL4 [86]. HOL4 is an interactive proof tool that can be used to formally prove properties of models of hardware and software. The verification is further described in Subsection 2.3.4.2.

Furthermore, the validation mechanism in the monitor can also be used to check other security policies than execution of signed code. Anti-virus analyzes can also be performed [86].

1.2 Problem Definition

Remaining work for PROSPER is to enable Linux to access the Internet on top of the hypervisor, and to prove that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed in such an environment. To enable Linux to access the Internet the hardware must be extended with a network interface controller (NIC) that allows Linux to send and receive messages. Such an extended system is the focus of this thesis and is shown in Figure 3.

The NIC introduces one problem with respect to the design and the verification described in the previous section. The design and the verification depends on the MMU in the CPU. Since the NIC has a Direct Memory Access (DMA) controller, the NIC can read and write memory independently of the configuration of the MMU and the page tables. If the hypervisor does not appropriately supervise the accesses Linux makes to the NIC registers, Linux could configure the NIC such that the NIC stores received messages in memory blocks that store page tables, executable code, or that are allocated to the hypervisor or the monitor. Such memory writes can potentially give the control of the system to Linux and/or enable execution of unsigned Linux code. The introduction of a NIC therefore invalidates the software design and the verification result described in the previous section.

The memory mapping request handlers must therefore be extended and complemented with a set of functions that ensure that Linux does not configure the NIC such that the NIC can break the separation or execution properties of the design. Those functions are referred to as the NIC register write request handlers. The remaining work for PROSPER therefore includes the following four tasks:

1. Extending the design of the memory mapping request handlers, and designing the NIC register write request handlers, such that they preserve the separation and execution properties.
2. Implementing the extended design of the memory mapping request handlers in the hypervisor and the monitor, and implementing the design of the NIC register write request handlers in the hypervisor.
3. Extending the hypervisor and Linux to enable Linux to access the Internet when Linux is executed on top of the hypervisor.
4. Formally verifying that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed.

This thesis focuses on the solutions to tasks one and three and partly tasks two and four. The following components have been used as the starting point:

- The development board BeagleBone Black (BBB) [31] which provides an ARMv7 CPU and a NIC.
- The hypervisor described in the previous section, but which has no network support.
- A paravirtualized Linux 3.10 kernel that can be executed on top of the hypervisor, but which has no support for Internet access.
- The design of the memory mapping request handlers [86] described in the previous section.

By using these components, the problem definition is to provide the following:

1. An extension of the design of the memory mapping request handlers and a design of the NIC register write request handlers, such that the hypervisor, the monitor and Linux are securely separated and only signed Linux code is executed.
2. An implementation of the NIC register write request handlers in the hypervisor.
3. An extension of the hypervisor and Linux such that Linux can access the Internet when Linux is executed on top of the hypervisor.
4. A proof plan that describes how it can be formally proved in HOL4 that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed.

   In order to make the result usable for PROSPER, the proof plan shall be based on the device model framework [85]. The device model framework is a model implemented in HOL4 and is partly developed by PROSPER. It is described in Subsection 2.4.1.

1.3 Method

The solutions to the four problems listed in the problem definition are described in this Section. To ensure that Linux does not configure the NIC to operate insecurely, the registers of the NIC are mapped as read-only in non-privileged mode. Since the NIC does not perform side effects on register reads, this mapping prevents Linux from directly configuring the NIC. When Linux attempts to write a NIC register, the CPU takes an exception which causes the NIC handling code to run. That code checks which NIC register Linux attempted to write with which value, and calls the NIC register write request handler that handles writes to that register with the value Linux attempted to write. The handler checks that the requested register write does not cause the NIC to write memory blocks that contain page tables, executable code, or that belong to the hypervisor or the monitor. If the register write satisfies these conditions, the register write is re-executed and otherwise blocked. The design of the memory mapping request handlers is extended with checks that ensure that blocks that the NIC can write are not used to store page tables or executable code. These checks performed by these two sets of handlers ensure that the hypervisor, the monitor and Linux are securely separated, and that executable blocks containing Linux code have signed contents.

The design of the NIC register write request handlers is specified in pseudocode to give it a clear structure and accurate meaning. To ease the implementation of the proof plan in HOL4, which depends on the design of these handlers, the pseudocode notation is defined to be similar to the HOL4 syntax.

The design of the NIC register write request handlers is implemented in the data abort exception handler of the hypervisor. The hypervisor can therefore call the handlers when Linux attempts to write a NIC register. Since the NIC register write request handlers are invoked by means of exceptions, it is not necessary to modify the NIC driver in the Linux kernel, which would be necessary if they were invoked by means of hypercalls.

Since the given paravirtualized Linux kernel was not configured with Internet support, its configuration had to be extended to include the necessary networking code. The networking code caused attempts to execute privileged operations related to cache and branch prediction management. These operations failed since Linux is executed in non-privileged mode. A practical solution was to implement these privileged operations in C as hypercalls in the hypervisor by following their corresponding assembly code implementation in the Linux kernel. Invocations to these hypercalls were then inserted at the corresponding locations in the Linux kernel.

To describe how it can be formally proved in HOL4 that only signed Linux code is executed, the proof plan must reason about the executions performed by the CPU and the NIC, and their interactions through accesses to the NIC registers and the memory. Those executions and interactions can be described by the device model framework, the models of the ARMv7 ISA and the MMU, and by a suitable model of the NIC. A suitable model of the NIC must match the I/O device interface of the framework, and it must describe all memory accesses that the NIC performs according to how the NIC device driver in Linux configures the NIC. Since no such model existed, one is provided in the same pseudocode notation that is used to specify the design of the NIC register write request handlers. The specification of the NIC model in pseudocode, the similarities between the syntaxes of the pseudocode and HOL4, and the interface of the NIC model make it straightforward to implement the NIC model in HOL4 and integrate it with the framework. The integration of the framework with a HOL4 implementation of the NIC model forms a HOL4 model that describes the executions of the relevant hardware that executes the hypervisor, the monitor and Linux. The proof plan reasons about that HOL4 model by means of the labeled transition system notation described in Subsection 2.1.2.

The proof plan is based on the simulation proof method: It is first proved that the software design ensures that only signed Linux code is executed, and then proved that this property can be transferred to the system executing the binary code of the hypervisor and the monitor. The proof plan consists two sets of lemmas, called the top- and sub-level lemmas, and a description of how the lemmas are applied to prove that only signed Linux code is executed. All lemmas are motivated by a description of why they hold, or if the lemmas involve unknown implementation aspects, it is described how the lemmas can be proved. The top-level lemmas are formulated as logical formulas and reflect the main ideas in the proof plan, while the sub-level lemmas support the top-level lemmas by considering deeper details.

This structure of the proof plan provides a holistic view of the proof approach while still taking details into account. The consistency between the notations of transition systems, the pseudocode and the logical formulas allow the proof plan to seamlessly reason about the hardware and the software design, which makes the proof plan relatively formal and precise. This precision together with the motivations of the lemmas make the proof plan believable. The reuse of the software design, and the formality and the soundness of the proof plan, make the proof plan especially suitable to guide an implementation of a formal proof in HOL4 of that only signed Linux code is executed.

1.4 Contributions and Conclusion

The solutions to the problems listed in the problem definition contribute in three respects to ensure that only signed Linux code is executed in an embedded system with network access:

- Software design: A specification that describes how the hypervisor and the monitor can ensure that only signed Linux code is executed.
- Implementation: An implementation of the networking aspects of the software design.
- Verification of correctness: A description of how it can be formally proved that the hypervisor and monitor implementation of the software design ensures that only signed Linux code is executed.

More specifically, the contributions are:

- A formal description of which conditions related to the NIC that the memory mapping request handlers must check in order to ensure that only signed Linux code is executed.
- A specification in pseudocode of the NIC register write request handlers that describes how the hypervisor can give Linux access to the NIC, such that the NIC cannot enable the execution of unsigned Linux code.
- An implementation of the NIC register write request handlers in the hypervisor.
- An extension of the hypervisor and the paravirtualized Linux kernel that enables Linux on BeagleBone Black to access the Internet.
- Benchmark results of network performance that illustrate the overhead of the implementations mentioned in the previous two bullets.
- A formal model of the NIC on BeagleBone Black that describes how the NIC accesses memory and asserts interrupts. The model describes these operations to the extent that they are used by the device driver of the NIC in Linux 3.10. The NIC is modeled as a transition system where each transition corresponds to one operation that accesses one field or byte of a NIC register or the memory. The model of the NIC is specified in functional pseudocode syntax to ease an implementation of it in HOL4.
- An identification of a set of HOL4 models which can be integrated with a HOL4 implementation of the NIC model to form a computer model in HOL4. That computer model describes the execution of a computer consisting of an ARMv7-A CPU, a memory and the NIC on BeagleBone Black. The resulting model is a transition system that describes the interaction between these hardware components, and between these hardware components and the software executed on top of it. The transition system consists of all possible interleavings of CPU and NIC transitions, where one CPU transition corresponds to the execution of one CPU instruction.
- A relatively formal and detailed pen-and-paper proof plan that describes how it can be formally verified that the binary code of the hypervisor and the monitor ensure that only signed Linux code is executed. This proof plan is based on the computer model mentioned in the previous bullet and the design of the memory mapping and NIC register write request handlers mentioned in the first two bullets. The verification of that only signed Linux code is executed does not only include Linux applications but also the kernel, device drivers and modules loaded on demand.

If the current implementation of the memory mapping request handlers is extended to consider the operation of the NIC, and the model of the NIC and the proof plan are implemented in HOL4, then an embedded system running Linux with Internet access is acquired, BeagleBone Black, such that, with high reliability, no malicious code in Linux is executed.

1.5 Outline of the Thesis

The rest of the thesis is structured as follows. Chapter 2 presents background material that allows an uninitiated reader to understand this thesis.

Chapters 3 through 6 describe the solutions to the four problems listed in the problem definition. Chapter 3 addresses the first problem: It describes the design of the extended memory mapping request handlers and the NIC register write request handlers. Chapter 4 addresses the second and the third problems: How the NIC register write request handlers are implemented and how Linux is given Internet access. Chapters 5 and 6 address the last problem: Chapter 5 describes the models used in the proof plan, and Chapter 6 presents the proof plan.

Chapter 7 motivates why the work described in Chapters 3 through 6 fulfills the requirements in the problem definition. Chapter 8 discusses from several perspectives which potential impacts the work described in this thesis can have.

Several appendices are also included. Their purpose is to present the fundamental details for the interested reader and guide future work. Appendix A describes the pseudocode notation used to specify the software design of the NIC register write request handlers and the model of the NIC. The pseudocode specifications of these handlers and the NIC model are partly included in Appendices B and C, respectively. Appendix B also contains a formal description of the extended memory mapping request handlers. To ease the understanding of the models used in the proof plan, Appendix D provides an example of an execution trace that is included in the model that describes how the hardware executes. Finally, Appendices E and F provide details that are part of the proof plan. Appendix E formally defines and describes the security invariant that is used in the proof plan to prove that the software design is secure. Appendix F motivates the sub-level lemmas.

2 Background

This chapter explains the notation used in this thesis, what formal verification is, the system environment Linux runs in, and earlier work that this thesis is based on. Related research is also described to put the work presented in this thesis in a wider perspective, and present tools and methods that might be helpful to formally prove that only signed Linux code is executed.

2.1 Notation

Appendix A describes the pseudocode notation that is used to specify the NIC register write request handlers in Appendix B and the NIC model in Appendix C. If the reader is not interested in those specifications, the notation described in this section is sufficient to understand the rest of the thesis, except for minor details that are explained when encountered for the first time.

2.1.1 Notation for Functions

Functions are defined by means of the definition symbol '$\overset{\text{def}}{=}$'.

Logical formulas are formed by ordinary logical and mathematical symbols with their classical meaning: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$, $\in$, $\subseteq$, $\cup$, $\times$. These symbols represent boolean negation, conjunction, disjunction and implication; for all and there exists quantifiers; element of, subset of, union of and direct product of sets. $\neg$ binds tighter than $\wedge$, which binds tighter than $\vee$, and which binds tighter than $\Rightarrow$. '[' and ']' are used as parentheses in logical formulas in order to ease the interpretation of their scope. '(' and ')' are mainly for function application.

A component of a tuple is referred to by the name of the tuple and the name of the component separated by a dot. For instance, if $a = (b, c)$, then the component b of a is referred to as $a.b$.

2.1.2 Notation for Labeled Transition Systems

The reasoning in the proof plan is based on a set of models, which describe the execution of hardware components. These models describe the executions as transition systems, and are therefore described in this thesis by means of a labeled transition system notation. In order to understand the models and the proof plan must the meaning of this labeled transition system notation be understood.

A labeled transition system consists of a four tuple $LTS = (S, IS, L, \delta)$, where:

- S is the set of states in the transition system.
- IS is the a set of initial states in the transition system.
- L is the set of labels that the transitions in the transition system have.
- $\delta \subseteq S \times L \times S$ is the transition relation of the transition system and consists of all transitions in the transition system. If $(s, l, t) \in \delta$, then it means that there is a transition in the transition system from the state s to the state t with the label l. This is also written as $s \rightarrow_l t$.

The transition relations used in this thesis are defined by means of transition rules. A transition rule has the following form:

$$\frac{P}{s \rightarrow_{l} t}.$$

The meaning of a transition rule is: If the premise *P* is true, then *s* can transition into the state *t* and the transition has the label *l*.

The following example clarifies the concept of a transition system and its notation. Consider the following set of transition rules, where each state consists of a pair of natural numbers:

- The following rule means that the state states (0, 0) and (0, 1) can transition into the states (0, 1) and (0, 2), respectively, where the labels of the transitions are *l_1*:

$$\frac{(a = 0 \wedge b = 0) \vee (a = 0 \wedge b = 1)}{(a, b) \rightarrow_{l\_1} (0, b + 1)}.$$

- This rule means that the state (0, 0) can transition into the state (1, 1) with the label of the transition being *l_2*:

$$\frac{a = 0 \wedge b = 0}{(a, b) \rightarrow_{l\_2} (1, 1)}.$$

These transition rules are used to define the transition relation δ in the transition system $LTS = (S, IS, L, \delta)$ as follows:

- $TS.S \overset{\text{def}}{=} \{(0, 0), (0, 1), (0, 2), (1, 1)\}$
- $TS.IS \overset{\text{def}}{=} \{(0, 0)\}$
- $TS.L \overset{\text{def}}{=} \{l\_1, l\_2\}$
- $TS.\delta \overset{\text{def}}{=} \{((0, 0), l\_1, (0, 1)), ((0, 1), l\_1, (0, 2)), ((0, 0), l\_2, (1, 1))\}$. These are the transition that are generated by the two transition rules listed above.

Two execution traces in this transition system are:

- $(0, 0) \rightarrow_{l\_1} (0, 1) \rightarrow_{l\_1} (0, 2)$, and
- $(0, 0) \rightarrow_{l\_2} (1, 1)$.

Also, this transition system is non-deterministic since there are two different transitions from the state (0, 0).

2.2 Formal Verification

To understand some design decisions and the verification approach of the system that hosts Linux, it is appropriate to start with an introduction to some common methods that are used to formally verify this kind of system. In this thesis, formal verification is referred to as the process of proving properties about abstract

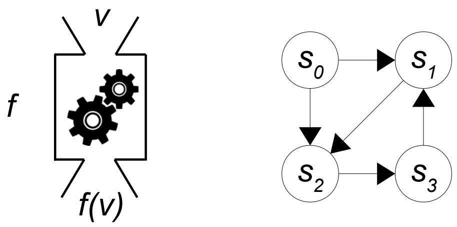


Figure 4: A graphical representation of two models. The model to the left represents a mathematical function, and the model to the right represents a transition system.

objects, called models, by means of mathematics, and where the proofs are machine-checked by a computer program. In the context of this thesis, the models describe behavior of software and hardware. Since mathematics is unambiguous and the proofs are checked by a computer program, the possibility of proving false properties is minimized. This gives a high reliability of that an implemented system actually has a proven property and which in principle proves the absence of bugs with respect to the proven property. Formal verification is therefore desirable to use in the development of critical applications.

Formal verification does not establish that a system has a proved property with complete certainty since there are verification gaps. For instance, discrepancies between the system and the model:

- If the system to analyze is complex and the model of it is hand-made, there is a significant risk for bugs in the model.
- The model might be too abstract and not reflect all the system behavior that the property depends on.
- Bugs in the implementation of the system might cause the system to not follow its specification, and where the specification has been used to construct the model.

It might therefore be possible to prove a property on the model that the system does not have. It is therefore critical that the model that describes the system is correct. The more accurate the model is, the more reliable is the formal proof to imply that the system has the proved property. An additional verification gap is the proof tool. Potential bugs in the proof tool might cause the proof tool to accept incorrect proofs.

Hardware and software can be modeled in several different ways. For instance, hardware components that just output a value given an input value can be modeled as ordinary mathematical functions, while executable systems that transition between states are suitably modeled as transition systems, as depicted in Figure 4. Models can be implemented either in the language that the proof tool uses to read its input, or in a separate specification language that is compiled to the input language of the proof tool.

Some commonly used tools to formally verify software or hardware are:

- Static analyzers: Commonly verify program properties. For instance, computational correctness, termination conditions, presence of unreachable code, and absence of division by zero.

  One sort of static analyzers verify annotated source code programs. The user annotates the program with logical formulas at certain control points, with the meaning that the property encoded by the formula holds at that control point for all program executions. The tool attempts to verify that all logical formulas hold at their corresponding control points. Verification of annotated source code does not need any user provided models, but verifies only that the program is correct with respect to the encoded formulas and the definition of the programming language. Hence, no compiler bugs are captured. Also, the operation of the hardware is not analyzed, which might be desirable for verification of programs that interact with I/O devices.
- Model checkers: Take as input a model, and a logical formula encoding a property. The tool attempts to automatically verify that the model satisfies the formula. A problem with model checkers is that complex models might require the verification process to traverse a large number of states, causing a potential state-space explosion. This potential state-space explosion may require memory that is not available in a regular computer/server, making this verification approach infeasible for complex systems. [36]
- Theorem provers: Allow users to construct a formal proof of that a property holds on a model. The tool checks that the user only performs sound steps in the construction of a proof. It also assists the user by automatically proving certain steps, providing tools that the user can use to organize the proof, and recording which steps that remain to be proved. Theorem provers are flexible in the sense that they allow the user to reason about complex systems with complex properties, but are time consuming in the sense that models must be constructed and users are required to perform a significant amount of work to construct a proof.

There are several books covering some of these tools and their usage [34-36].

2.3 The System

The system that hosts Linux consists of an ARMv7 CPU, a memory, a NIC, a hypervisor and a monitor. These hardware and software components are described in the following subsections.

2.3.1 ARMv7 Instruction Set Architecture

The CPU implements the 32-bit ARMv7 instruction set architecture (specifically, ARMv7-A [33]). The CPU executes in either privilege level zero (PL0) or privilege level one (PL1): PL0 for non-privileged application software that must not have access to system resources, and PL1 for privileged operating system (OS) software that controls system resources. There are seven execution modes, called usr, fiq, irq, svc, und, abt and sys. usr mode is the execution mode for applications and executes in PL0, while the other modes are the execution modes for the OS and execute in PL1. The CPU transitions from usr mode to fiq, irq, svc, und or abt mode only when an exception is taken, and sys mode is only entered manually by the OS from fiq, irq, svc, und or abt mode.

When an exception is taken, the CPU sets the program counter to a preconfigured value and enters fiq, irq, svc, und or abt mode. The new value of the program counter addresses a memory location where the OS is stored, causing the OS to be executed and handle the exception. When the OS has handled the exception, it instructs the CPU to load its registers with the state of the next application to execute and to transition to usr mode. The execution of that application then continues.

The exceptions that can occur on an ARMv7 CPU are:

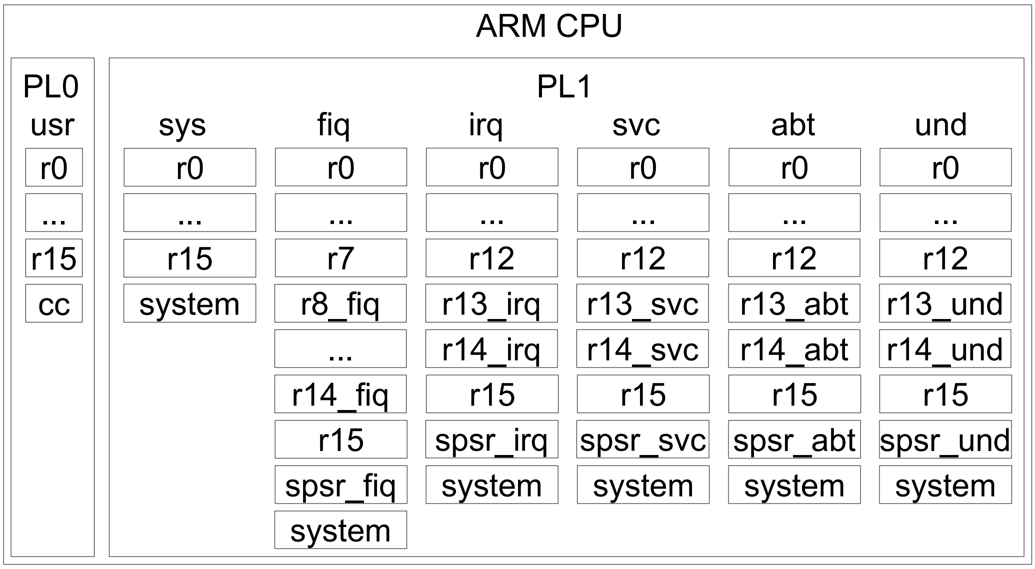
- FIQ and IRQ interrupts: Cause the CPU to enter fiq and irq modes, respectively. These exceptions are raised by I/O devices. fiq mode is entered when high-priority devices assert interrupts (Fast Interrupt reQuest), and irq mode is entered when low-priority devices assert interrupts (Interrupt ReQuest). In the system setting that this thesis focuses on, only the NIC can assert interrupts, and in particular IRQ interrupts.
- Supervisor call exceptions: Cause the CPU to enter svc mode. These exceptions are raised when executions of application software execute a supervisor call instruction to invoke a system call of the OS.
- Undefined instruction exceptions: Cause the CPU to enter und mode. These exceptions are raised for instance when the CPU attempts to execute an undefined instruction.
- Prefetch and data abort exceptions: Cause the CPU to enter abt mode. These exceptions are raised for instance when the CPU attempts to fetch an instruction or access data at a memory location to which no access is currently allowed, respectively.

  The OS can distinguish between prefetch and data abort exceptions since the program counter is set to 0xFFFF000C for prefetch abort exceptions and to 0xFFFF0010 for data abort exceptions.

The CPU has 16 general-purpose registers, which include the stack pointer, the link register (used to store the return address for function calls), and the program counter. Some of these registers are banked between several execution modes, meaning that some execution modes have their own copies of these registers. The CPU contains also several system registers, some of which configure the hardware. Figure 5 shows the relation between privilege levels, execution modes and registers in an ARM CPU.

The relevant system registers in this thesis are:

- CPSR: The current program status register affects the execution of the CPU. Among other things, CPSR contains condition code flags that determine the outcome of conditional branch instructions, which execution mode the CPU is in, and whether interrupts are enabled or not. Bits seven and six of CPSR determine whether the CPU responds to IRQ and FIQ interrupts, respectively. If these bits are set, the CPU ignores the



ARM CPU						
PL0	PL1					
usr	sys	fiq	irq	svc	abt	und
r0	r0	r0	r0	r0	r0	r0
...	...	...	...	...	...	...
r15	r15	r7	r12	r12	r12	r12
cc	system	r8_fiq	r13_irq	r13_svc	r13_abt	r13_und
		...	r14_irq	r14_svc	r14_abt	r14_und
		r14_fiq	r15	r15	r15	r15
		r15	spsr_irq	spsr_svc	spsr_abt	spsr_und
		spsr_fiq	system	system	system	system
		system				

Figure 5: The relation between privilege levels, execution modes and registers in an ARM CPU. usr mode has only access to 16 general-purpose registers and the condition code flags of the CPSR register. Each privileged execution mode has access to most of the registers that are accessible in usr mode, but with some of those registers replaced by registers that are only accessible in that privileged execution mode. The registers that are only accessible in a specific privileged execution mode have their names appended by an underscore followed by the name of that execution mode. The registers r13, r14 and r15 of each execution mode are the stack pointer, link register and program counter, respectively. All execution modes that can be entered by an exception have their own copy of the SPSR register. Also, only the privileged execution modes have access to the system registers. Examples of system registers are CPSR, TTBR0 and DACR, which affect the operation of the hardware, and SPSR and DFAR, which do not affect the operation of the hardware.

corresponding interrupt. That is, if the bits are set, the interrupts are masked.

- SPSR: The saved program status registers are used by the CPU to store the value of the CPSR register when the CPU takes an exception. Each execution mode that can be entered by an exception has its own SPSR register (see Figure 5). Immediately before the CPU takes an exception, the CPU stores the value that CPSR contains, into the SPSR register that is accessible to the execution mode that is entered after the exception has been taken. For instance, if an IRQ interrupt occurs, immediately before the CPU takes that exception, the CPU stores the value of CPSR in SPSR_irq.

  By reading SPSR, the OS can restore the value of CPSR after an exception has been handled to the value CPSR contained before an exception was taken. It is necessary for the OS to be able to restore CPSR since CPSR affects the execution of applications and the OS might modify CPSR. For instance, when the CPU is executing an if-then-else statement of an

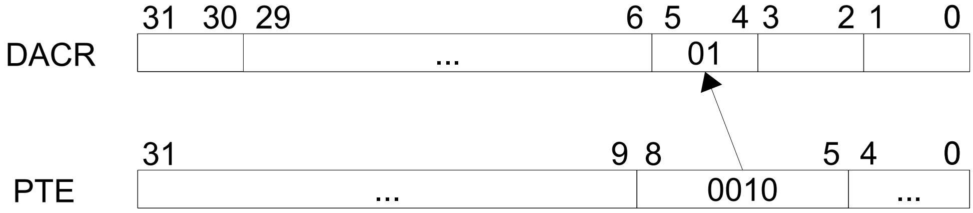


Figure 6: The relation between a first-level page table entry (PTE) and the DACR register. Bits 8 to 5 in a first-level page table entry contains the index value to the field of the DACR register that is used to determine how access permissions are computed. The binary index value of the shown page table entry is 0010, which identifies the third field of the DACR register. That third field of the DACR register contains the binary value 01. Hence, the access permissions of the virtual addresses mapped by this page table entry are specified by the page tables. If bits 5 and 4 of DACR contained 00, no access would be allowed to those virtual addresses, and if bits 5 and 4 contained 11, all accesses to those virtual addresses would be allowed. If this page table entry maps 1 MB of memory, the access permissions are encoded in bits 15, 11 and 10 of this page table entry. If this page table entry points to a second-level page table, the access permissions are encoded in the second-level page table entry that maps the accessed virtual address.

application, which might affect the condition code flags, the CPU can take an exception. When the OS is executed to handle the exception, the CPU might execute an if-then-else statement of the OS, causing the condition code flags to change. CPSR must therefore be restored to allow the continued execution of the application to be correct.

- TTBR0: The translation table base register zero contains the physical address of the memory location that contains the first entry of the first-level page table. This register is used by the MMU and its role is described below.
- DACR: The domain access control register is used in combination with page table entries to compute access permissions of virtual addresses. It consists of 16 fields of two bits each, where each field is identified by an index value from zero to fifteen. Each first-level page table entry contains one such index value. The two bits in the field of the DACR register with that index determine how the access permissions are computed for virtual addresses that are mapped by that page table entry (see Figure 6; the page tables are described below). The meaning of the bits in the fields of the DACR register is:
  - 0b00: No access.
  - 0b01: Access permissions are determined by the page tables.
  - 0b10: Invalid encoding (unused).
  - 0b11: Allowed access.

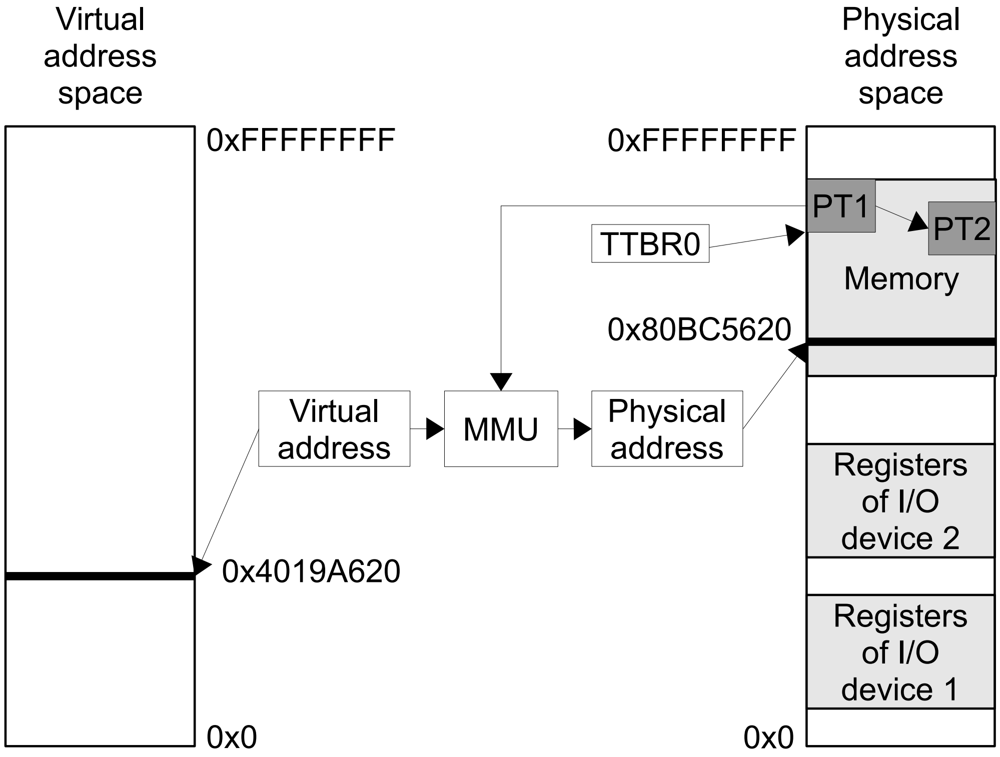


Figure 7: The relationship between the virtual address space and the physical address space. The CPU uses virtual addresses to access memory and registers of I/O devices, while the latter are located in the physical address space. In this example, the physical address space contains memory and registers of two I/O devices, leaving the rest of the physical address space unused. The MMU reads a set of page tables in memory to translate a virtual address to a physical address. Two page tables, referred to as PT1 and PT2, are shown in the memory. When the MMU translates a virtual address, the MMU first reads the first-level page table whose physical base address is located in TTBR0. In this example, PT1 is the first-level page table. To translate the virtual address 0x4019A620, PT1 specifies that the MMU shall read PT2 (hence the arrow from PT1 to PT2), which is a second-level page table. Since PT1 and PT2 specify that the virtual address 0x4019A620 shall be mapped to the physical address 0x80BC5620, the MMU outputs the physical address 0x80BC5620 when given the virtual address 0x4019A620. Hence, the CPU accesses the physical address 0x80BC5620 when specifying the virtual address 0x4019A620. Since the physical address 0x80BC5620 identifies a memory location, the CPU accesses memory when specifying the virtual address 0x4019A620.

- DFAR: The data fault address register contains the virtual address that the CPU attempted to access but which caused a data abort exception. For instance, if the CPU attempted to write a non-writable virtual memory location at virtual address 0xFA400000, then DFAR contains 0xFA400000.

The CPU accesses a memory location or a register of an I/O device by specifying a virtual address. Given a virtual address, the MMU, a part of the CPU, performs a translation table walk that traverses one or two page tables located in memory. The page tables specify how the MMU shall compute the access permissions (read, write or execute) and the physical address of a given virtual address. If the computed access permissions are compatible with the requested access, the CPU accesses the memory location or the register of an I/O device located at the physical address, if the hardware is configured correctly. The relationship between the virtual addresses, the MMU, the page tables and the physical addresses is illustrated in Figure 7.

The page tables are organized into two levels as follows (there are also other organizations but they are not relevant for this thesis):

1. First-level page tables contain 1024 entries. Each entry is either free, maps 1 MB of virtual memory to 1 MB of physical memory (256 consecutive 4 kB memory blocks), or points to a second-level page table.
2. Second-level page tables contain 256 entries. Each entry is either free or maps 4 kB of virtual memory to 4 kB of physical memory.

Given a virtual address, the MMU performs a translation table walk as follows (as is also illustrated in Figure 8). The MMU first reads the TTBR0 register to find the first-level page table. Bits 29 to 20 of the virtual address are then used as an index to that page table to identify the first-level page table entry that maps the given virtual address (bits 31 and 30 are handled in a special way that can be skipped in this description):

- If the entry is free, a data abort exception is raised. A free entry means that the virtual memory region that the virtual address belongs to is unmapped and inaccessible.
- If the entry maps a 1 MB memory region, the entry contains the physical base address of that memory region, an index to a field of the DACR register, and the access permissions of the given virtual address. The MMU adds bits 19 to 0 of the virtual address to the physical base address of the 1 MB memory region to compute the physical address that the given virtual address is mapped to.
- If the entry points to a second-level page table, it contains the physical address of the first entry of that second-level page table, and an index to a field of the DACR register. Bits 19 to 12 of the virtual address are then used as an index to this second-level page table to identify the second-level page table entry that maps the given virtual address. If the second-level page table entry is free, a data abort exception is raised.

  Otherwise the second-level page table entry maps a 4 kB memory region, and contains the physical base address of that memory region together with the access permissions of the given virtual address. The MMU adds bits 11 to 0 of the virtual address to the physical base address of the 4 kB memory region to compute the physical address of the given virtual address.

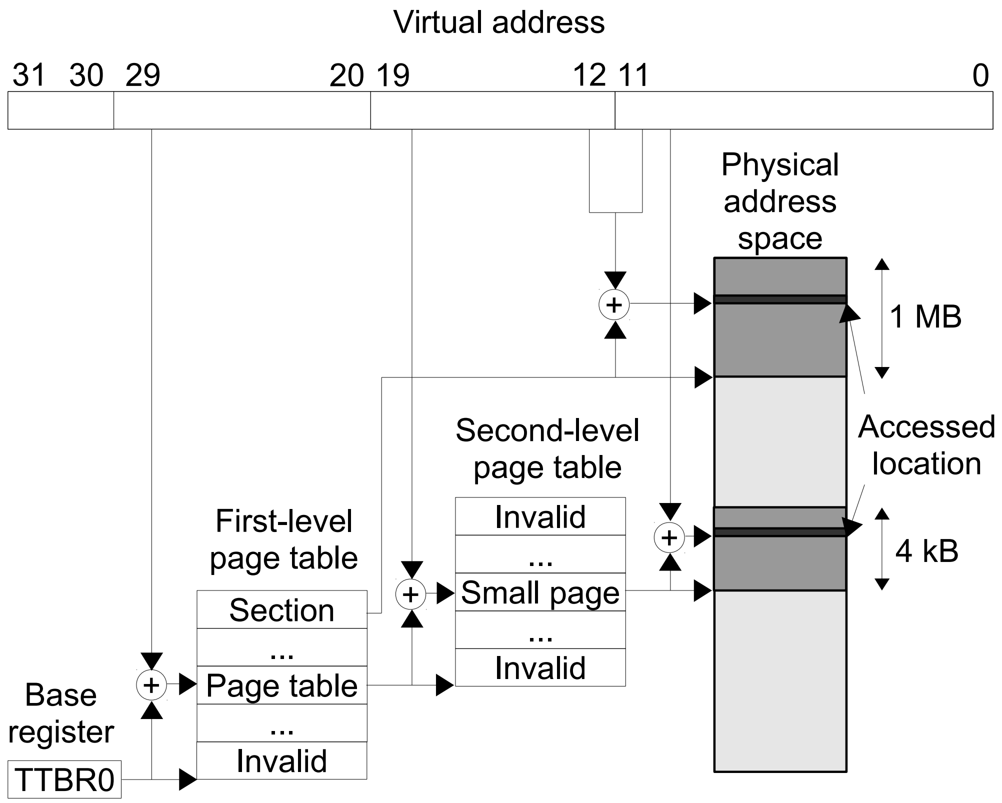


Figure 8: How the MMU performs a translation table walk to map a virtual address to a physical address. The MMU uses the TTBR0 register to find the first-level page table. A first-level page table consists of 1024 entries where, in this example, the first entry is free (such an entry is called Invalid by ARM), some intermediate entry points to a second-level page table, and the last entry points to a memory region of 1 MB (called Section). The second-level page table consists of 256 entries, where the first and last entries are free and one intermediate entry points to a memory region of 4 kB (called Small page). To map a virtual address, the MMU traverses either only one first-level page table, or one first-level page table and then one second-level page table. The physical address of the first-level page table entry that maps a given virtual address is computed by adding bits 29 to 20 of the virtual address to the contents of TTBR0. If the first-level page table entry points to a 1 MB memory region, the physical address that the virtual address is mapped to is computed by adding bits 19 to 0 of the virtual address to the physical address of that memory region. The CPU then accesses the location in the physical address space with the computed physical address, if allowed by the access permissions that are computed during the translation table walk. If the first-level page table entry points to a second-level page table, the physical address of the second-level page table entry that maps the given virtual address is computed by adding bits 19 to 12 of the virtual address to the physical address of the second-level page table. The second-level page table entry points to a 4 kB memory region. The physical address that the virtual address is mapped to is computed by adding bits 11 to 0 of the virtual address to the physical address of the 4 kB memory region. The CPU accesses the location in the physical address space identified by the computed physical address, if the access permissions allow.

The translation table walk allows the MMU to identify the physical address and the access permissions of a given virtual address. The granularity of the access permissions is with respect to the privilege level of the CPU (PL0 or PL1), where the access permissions are readable, writable and executable.

If the value of the field of the DACR register and the access permissions identified by the page table entries are compatible with the current privilege level of the CPU, the CPU accesses the location in the physical address space that is identified by the computed physical address. If a virtual address identifies an instruction, the page table entry mapping that virtual address must map the virtual address as both readable and executable in order for the CPU to execute the instruction.

If the access permissions are violated, a prefetch or data abort exception is raised, causing the CPU to take the corresponding exception and execute privileged software to handle it. When instruction fetches (to execute an instruction) violate access permissions, prefetch abort exceptions are raised, and when data memory accesses violate access permissions, data abort exceptions are raised.

2.3.2 Network Interface Controller

The purpose of the NIC [32] is to transmit and receive messages, which are called frames. Software configures the NIC to transmit and receive frames by performing the following steps:

1. Resetting the DMA hardware and initializing certain registers.
2. Initializing the buffer descriptors that inform the NIC where in memory it shall fetch and store frames.
3. Writing the registers that start transmission and enable reception.
4. Acknowledging interrupts that the NIC generates when it has completed transmission or reception of frames.
5. Tearing down transmission and reception when the NIC is to be shut down. This is normally done when the computer is shut down or put into sleep mode.

There are a number of registers that affect which memory accesses the NIC performs. All of those registers must be protected by the hypervisor to prevent Linux from taking control of the system and the CPU from executing unsigned Linux code. Linux uses nine of the NIC registers that affect which memory accesses the NIC performs. The location of eight of those registers in the physical address space is shown in Figure 9. The ninth register, RX_BUFFER_OFFSET, is only written by Linux during initialization, and is described in Section C.1. The following five subsections describe the five steps listed above and how the eight registers shown in Figure 9 are used.

2.3.2.1 Resetting DMA Hardware

Before the NIC starts transmission and reception must the DMA hardware be reset. The NIC specification [32] does not specify what this reset operation does, but by observing the behavior on BBB, it seems that the reset operation only puts the NIC in an inactive state without doing anything else (e.g. resetting some registers). The

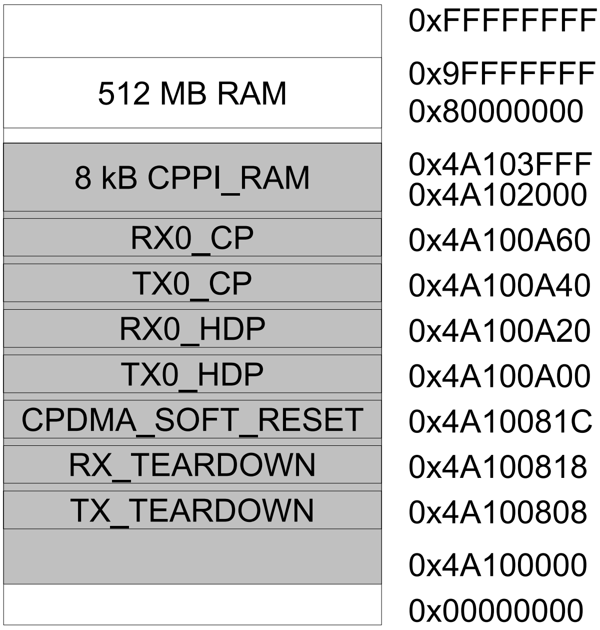


Figure 9: The location of the NIC registers and RAM in the physical address space of BeagleBone Black. The NIC registers constitute 16 kB of the physical address space and starts at address 0x4A100000. This is the shaded region. The RAM constitutes 512 MB of the physical address space and starts at address 0x80000000. Eight of the nine NIC registers that Linux uses and that affect which memory accesses the NIC performs are shown. The ninth register is RX_BUFFER_OFFSET and is described in Section C.1.

DMA hardware is reset by software by setting the least significant bit of the CPDMA_SOFT_RESET register. When the reset operation is complete, the NIC clears this bit. The reset register is relevant for memory accesses because, according to the NIC specification, the DMA hardware must be reset before transmission and reception start, both of which affect memory accesses.

2.3.2.2 Initializing Buffer Descriptors

Before transmission and reception can start, the NIC must know where the data buffers used to store frames are located. That information is given to the NIC by means of a set of buffer descriptors. Each buffer descriptor consists of four 32-bit words and is stored in an 8 kB memory in the NIC called CPPI_RAM. A buffer descriptor consists of several fields where the most important are (Figure 10 illustrates the use of these fields and their roles by means of an example):

- Next Descriptor Pointer: Points to another buffer descriptor to allow the construction of transmit and receive queues. If it has the value zero, the buffer descriptor is the last one in the queue.
- Buffer Pointer: Points to a memory location to specify where the addressed data buffer of the buffer descriptor starts.
- Buffer Length: Specifies the size of the addressed data buffer in bytes.

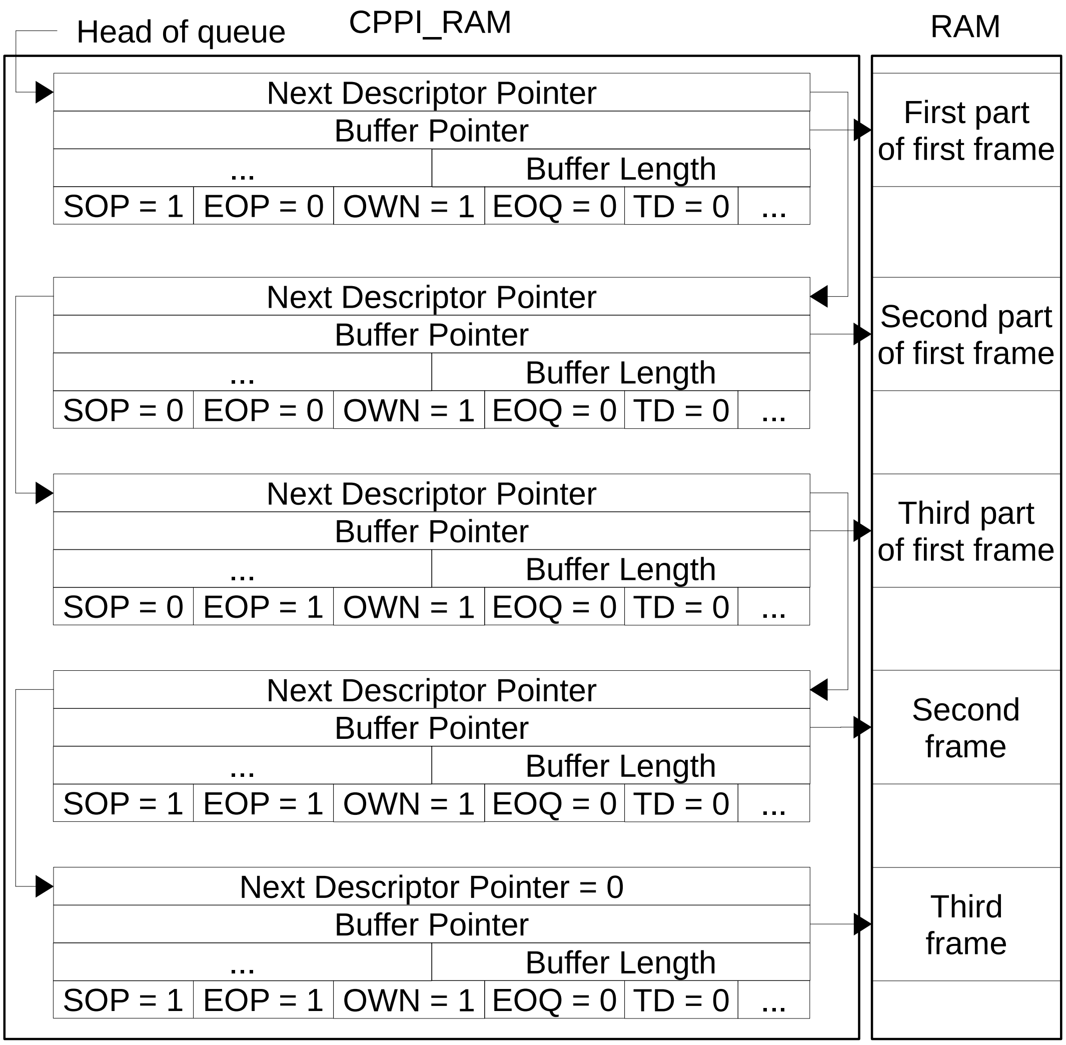


Figure 10: Five initialized buffer descriptors chained together into a transmission queue ready to be given to the NIC. The first three buffer descriptors specify how the NIC shall fetch the first frame from memory. The first buffer descriptor addresses the first part of the first frame since it is at the head of the queue and its SOP bit is equal to one. The second buffer descriptor addresses the second part of the first frame since it follows the first buffer descriptor in the queue and its SOP and EOP bits are equal to zero. The third buffer descriptor addresses the last part of the first frame since it follows the second buffer descriptor in the queue and its EOP bit is equal to one. The following two buffer descriptors specify how the NIC shall fetch the next two frames. Since those two buffer descriptors identify complete frames, both their SOP and EOP bits are set. To allow software to know when the NIC (i) has released the buffer descriptors addressing a frame, (ii) has processed the queue, or (iii) teared down the queue, the buffer descriptors addressing the first part of a frame have their OWN bits set to one, and their EOQ and TD bits set to zero, respectively. The buffer length field specifies the size in bytes of the data buffer the buffer pointer field identifies. The next descriptor pointer field of the fifth buffer descriptor is equal to zero since it is the last buffer descriptor in the queue.

- NIC management fields. The most important are:
    - Start/End of Packet (SOP/EOP) bits: Set to one if the buffer descriptor addresses the first/last part of a frame. For transmission, this field is set by software and for reception, it is set by the NIC. This field allows both software and the NIC to know which data buffers that belong to a certain frame.
    - The ownership (OWN) bit: Set to one by software in the buffer descriptor addressing the first part of a frame to transmit (SOP buffer descriptor), and in all buffer descriptors in a receive queue. When the NIC has completed transmission or reception of a frame addressed by a set of buffer descriptors, the NIC clears this bit in the (SOP) buffer descriptor addressing the first part of that frame. The value of this bit informs software of when it can reuse the buffer descriptors addressing a certain frame, and when a frame has been transmitted or received.
    - End of Queue (EOQ) bit: Set to one by the NIC in the last buffer descriptor in a queue, when the NIC has processed all buffer descriptors in that queue. This field allows software to know if the NIC has processed a complete queue.
    - Teardown completion (TD) bit: Set to one by the NIC in the first unused buffer descriptor in a queue when that queue has been teared down. This field allows software to know if the NIC has finished the tear down operation of a queue.

Before a frame or a set of frames are transmitted must the buffer descriptors addressing them be chained together to form a queue, and likewise to enable reception must a set of buffer descriptors addressing a set of free data buffers be chained together to form a receive queue. A transmission queue that has been initialized by software and that is ready to be given to the NIC is shown in Figure 10.

2.3.2.3 Initiating Transmission and Reception

When the buffer descriptors addressing a frame or a set of frames are initialized and chained together to a queue, software can make the NIC transmit those frames by writing the physical address of the first buffer descriptor in that queue to the TX0_HDP register (Head Descriptor Pointer). For instance, if the queue in Figure 10 is to be transmitted and the first buffer descriptor, identified by the label "Head of queue", is located at physical address 0x4A102000, then TX0_HDP is written to 0x4A102000. When TX0_HDP has been written, the NIC processes all buffer descriptors in the queue and transmits the frames addressed by them, until the end of the queue is encountered, at which point the NIC sets TX0_HDP to zero.

Reception works similarly by writing RX0_HDP with the physical address of a receive queue but with the meaning that the buffer descriptors address memory that can be used to store received frames. Received frames cannot be stored in memory if all buffer descriptors in the receive queue have been consumed by the NIC.

There are eight transmission DMA channels and eight reception DMA channels where each one processes its own queue of buffer descriptors. Linux only uses one transmission DMA channel and one reception DMA channel, both of which have the index zero (thereby the zeros in the register names TX0_HDP and RX0_HDP). Also, the HDP registers must be initialized to zero after a reset operation and before they are used to transfer frames.

2.3.2.4 Acknowledging Interrupts

When a DMA channel has completed the transfer of a frame, the NIC writes the CP register (Completion Pointer) of that DMA channel with the physical address of the buffer descriptor that addresses the last part of the transferred frame. For instance, when the first frame in Figure 10 has completed transmission, and if its third buffer descriptor is located at physical address 0x4A102020, then the CP register of the corresponding DMA channel is set to 0x4A102020. These writes performed by the NIC asserts a frame transmission or reception completion interrupt, depending on which CP register is written.

Software acknowledges such interrupts by writing the corresponding DMA channel's CP register with its current value, at which time the NIC deasserts the interrupt. Since Linux only uses DMA channels zero, Linux only uses the TX0_CP and RX0_CP registers to acknowledge interrupts. The CP registers are also used for teardown interrupts (see next subsection). (The NIC can generate other interrupts as well but they are not described since Linux does not use them.) Also, the CP registers should be initialized to zero after a DMA hardware reset operation.

2.3.2.5 Tearing down Transmission and Reception

Software can make the NIC abort the transmission of a queue or disable the reception to a queue by writing the TX_TEARDOWN and RX_TEARDOWN registers, respectively. When software writes these two registers with the index of the DMA channel to abort, the NIC initiates a tear down operation on the corresponding DMA channel. When such an operation is initiated, the NIC first completes the transmission or reception of the currently processed frame. Then, the NIC writes the HDP register of the teared down DMA channel with the value zero, sets the TD bit in the first unused buffer descriptor, and writes 0xFFFFFFFC to the DMA channel's CP register to generate a teardown interrupt. For instance, assume that the buffer descriptor queue in Figure 10 is given to transmission DMA channel zero and a teardown command is initiated while the NIC transmits the first frame of that queue. After the NIC has transmitted that frame, the NIC sets TX0_HDP to zero, the TD bit in the fourth buffer descriptor to one, and TX0_CP to 0xFFFFFFFC. Software is expected to acknowledge teardown interrupts by writing 0xFFFFFFFC to the corresponding DMA channel's CP register. In the previous example, software is expected to write 0xFFFFFFFC to the TX0_CP register.

The teardown registers are relevant for memory accesses because teardown operations affect which memory the NIC accesses, since the unused buffer descriptors in the teared down DMA channel will not be processed by the NIC. The CP registers are not critical for memory accesses but are related to interrupts.

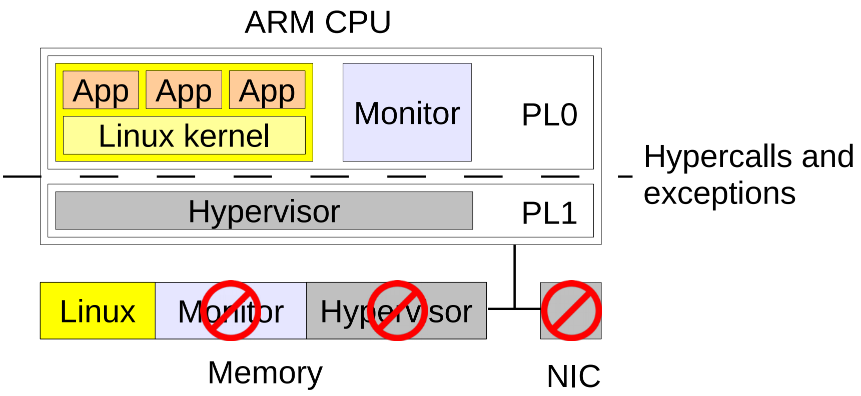


Figure 11: Linux and the monitor (a bare-metal application) running in non-privileged mode on top of the hypervisor in a system with an ARM CPU connected to a NIC. Linux is currently executed and the hypervisor has configured the system such that Linux can only access its own memory. The guests can communicate with the hypervisor through hypercalls and exceptions.

2.3.3 Hypervisors

A hypervisor runs directly on the hardware. On top of the hypervisor several guests can run independently of each other. A guest can, for instance, be an operating system with applications running on top of it, or a bare-metal application that does not run on top of an operating system, as is the case with the monitor used in the work described in this thesis. The purpose of the hypervisor is to allow several software systems to share the same hardware without interfering with each other, except for intended communication: The guests can only communicate with each other via dedicated communication channels provided by the hypervisor.

To prevent guests from interfering with each other in illegal ways, the system resources must be protected by the hypervisor. System resources are system registers in the CPU, page tables and I/O devices. In the ARM context, the system registers in the CPU are protected from the guests by making the hypervisor be executed in PL1 and the guests in PL0. The page tables and the I/O devices are protected by configuring the page tables such that the execution of the current guest can only access the memory of that guest and not write the page tables or configure the I/O devices. A system configuration of this kind is illustrated in Figure 11 for the system that this thesis is concerned with when Linux is executed.

Giving the guests arbitrary access to page tables or I/O devices must be prevented since otherwise the guests could:

- Change the access permissions in the page tables to get control of the system.
- Configure the devices such that they enter unknown states, making both hardware and software operate incorrectly.
- Configure the devices, if they have DMA support, to read or write the memory belonging to the hypervisor or other guests. These memory

accesses could give a guest access to sensitive data or result in that the guest takes control of the system.

On ARM, operating systems are intended to be executed in PL1 and applications are intended to be executed in PL0. An operating system can therefore isolate itself from the applications. Since the hypervisor must run in PL1, the guest operating systems must be modified in order to be executed correctly in PL0. Such a modified operating system is said to be paravirtualized, as is the case for Linux that this thesis deals with. Having a paravirtualized operating system running on top of a hypervisor normally implies that the hypervisor provides a software interface that the guests are designed to be executed on top of. This interface consists of a set of functions, called hypercalls.

When a guest needs to access a protected system resource, the guest invokes the hypervisor with parameters containing information of both which hypercall the guest wants the hypervisor to invoke and which operations the guest wants the hypercall to perform. On ARM, the execution of a guest invokes the hypervisor by means of a supervisor call instruction, the execution of which raises a supervisor call exception. That exception causes the CPU to transition from usr mode to svc mode and set the program counter to point to a location in the memory where the hypervisor is stored. From there, the hypervisor is executed and investigates the provided parameters in order to invoke the correct hypercall. The hypercall then checks if the requested operations are secure, and executes them only if they are.

All communication between the hypervisor and the guests must not necessarily occur through hypercalls. Another method is to use exceptions. Communication through exceptions is suitable in the context of I/O devices, since there is no need to modify the device drivers in the guests. By configuring the page tables to securely map the device registers, access violations will occur when a guest attempts to access a device register. These violations cause the CPU to take an exception and execute the hypervisor. The hypervisor investigates which device register the guest attempted to access, and if it is a write, which value the guest attempted to write. The hypervisor then decides whether the attempted operation is secure, and if so re-executes it. Otherwise is the operation blocked. For instance, on ARM, a hypervisor can read the DFAR register and the banked link register (which stores the contents of the program counter before an exception occurred) to find out which virtual address the guest attempted to access and which instruction of the guest that caused the exception. From that information, the hypervisor can identify which device register the guest attempted to access, if the access was a read or write, and if it was a write which value the guest attempted to write.

Letting guest operating systems be executed in non-privileged mode along with their applications raises the concern of the isolation of the guest operating system. Guest operating systems can be isolated from their applications as follows:

- The guest operating system configures its memory mapping such that the execution of the current application can only access the memory of that application.
- If a guest requests to access a critical resource, the hypervisor checks that the request is issued from the guest operating system and not from an

application. Hence, applications cannot fool the hypervisor that the guest operating system is issuing a request to access a critical resource.

These two operations imply that an application initially can only access non-privileged registers and its own memory, and that this property is preserved during the execution of the application. Since the guest operating system is not storing any data in non-privileged registers while applications are executed, the guest operating system is isolated from the applications.

2.3.4 Hypervisor and Monitor

The previous subsections 2.3.1 through 2.3.3 have described the ARMv7 ISA, the NIC on BBB, hypervisors and software running on top of hypervisors from a general perspective. This subsection, in contrast, describes the specifics of the original hypervisor and monitor which the work presented in this thesis had as a starting point.

2.3.4.1 Hypervisor

As described in Section 1.1, the hypervisor is executed in privileged mode and Linux and the monitor in non-privileged mode. All three software components have their own statically allocated physical memory regions. The page tables are located in the physical memory region allocated to Linux and are always mapped as read-only in non-privileged mode. The page tables are configured such that Linux can only access its own physical memory region, and as Linux has configured the page tables that map that physical memory region. The monitor has read access to the physical memory region allocated to Linux in order to be able to inspect whether newly mapped memory blocks contain signed code or not. The monitor can also read a few critical data structures located in the physical memory region allocated to the hypervisor in order to decide whether new page table entries are secure or not (as will be seen shortly). Linux, the hypervisor and the monitor are therefore not completely isolated, but sufficiently separated such that only the hypervisor can access and modify critical system resources.

The exceptions that can occur are handled by the hypervisor as follows:

- FIQ interrupts: Cannot occur since no I/O devices are configured as high-priority devices.
- IRQ interrupts: Are raised when the timer or the UART device asserts an interrupt. These devices are not critical to ensure that only signed Linux code is executed, since they cannot access memory. They are therefore ignored. However, only Linux uses these devices. IRQ interrupts are therefore directly forwarded by the hypervisor to the Linux kernel by setting the program counter to the address of the IRQ interrupt exception handler in the Linux kernel.
- Supervisor call exceptions: The hypervisor has a data structure that records whether it is the Linux kernel that is executed (virtual privileged mode) or if an application is executed (virtual non-privileged mode). If this exception occurs when Linux is executed in virtual non-privileged mode, it means that an application invoked a system call. The hypervisor therefore sets the

program counter to the virtual address of the system call handler routine in the Linux kernel.

If this exception occurs when Linux is executed in virtual privileged mode, it means that the Linux kernel invoked a hypercall. The hypercalls provided by the hypervisor handle among a few other tasks: masking of interrupts (I flag of CPSR), cache management (flush and invalidate cache blocks), restoring CPU registers after the Linux kernel has handled an exception, and memory mapping requests.

- Undefined instruction exceptions: Can only occur when Linux is executed since the hypervisor and the monitor are trusted. For instance, the code of the hypervisor and the monitor do not contain incorrectly encoded instructions. These exceptions are therefore forwarded to the undefined instruction exception handler in the Linux kernel.
- Prefetch abort exceptions: Are either forwarded to the prefetch abort exception handler in the Linux kernel or handled by the hypervisor. The hypervisor uses these exceptions to change access permissions for memory blocks that Linux requested to map as both writable and executable but where execute permission was revoked, since no blocks are allowed to be both writable and executable. In these cases, the hypervisor changes the access permissions from writable to executable if the accessed block contains signed code, and sets the program counter to the instruction that caused the exception. The faulting instruction in Linux memory is then re-executed, which will this time succeed.
- Data abort exceptions: Are either forwarded to the data abort exception handler in the Linux kernel or handled by the hypervisor. The hypervisor uses these exceptions in a similar way as prefetch abort exceptions but where write permission to the accessed block was revoked.

In the proof plan in Chapter 6 it is necessary to know whether it is the hypervisor, the monitor or Linux that is executed by the CPU in a given state. The executed software component is determined by the current values of the CPSR and DACR registers as follows. The page table entries that map the physical memory regions allocated to the hypervisor, the monitor and Linux have DACR field indexes zero, one and two, respectively. Bits five and four of DACR, denoted as DACR[5:4], specify how access permissions to the physical memory region allocated to Linux are computed. If CPSR specifies the CPU to be in non-privileged mode and DACR[5:4] = 0b01, then Linux is executed and is accessing its memory as it has configured its page table entries. If the CPU is in non-privileged mode and DACR[5:4] = 0b11, then the monitor is executed with access to the physical memory region allocated to Linux. Hence, the monitor can check whether code in the physical memory region allocated to Linux is signed. If the CPU is in privileged mode, the hypervisor is executed.

2.3.4.2 Memory Mapping Request Handlers and Monitor

The memory mapping request handlers operate as follows [86]. Since the physical address space is 32 bits wide, each 4 kB physical memory block can be identified by a block index consisting of 20 bits. The hypervisor maintains three data structures to record the state of each such block, and which take a block index as argument. The first data structure τ records the type of each block: *L1*, *L2* or *D*. Blocks of type *L1* and *L2* contain first- and second-level page tables, respectively. Blocks of type *D* contain all other data, including code. The other two data structures, ρ_{wt} and ρ_{ex}, record the number of page table entries in page tables located in *L1* and *L2* blocks that map a given block as writable and executable, respectively.

Recall that a memory mapping request handler is implemented by both the hypervisor and the monitor. When a memory mapping request handler is invoked, the hypervisor performs two checks. One check sees if the request specifies a page table modification that gives Linux access to physical memory allocated to the hypervisor or the monitor, or writable access to a block of type *L1* or *L2*. The other check sees if the request specifies a page table modification such that a first-level entry in a page table in an *L1* block refers to a second-level page table that is not in an *L2* block. If the request satisfies any of these two checks, the hypervisor rejects it. Otherwise the request is forwarded to the monitor. The monitor validates the request and returns its answer to the hypervisor. If the monitor accepts the request, the hypervisor executes it and otherwise rejects it.

The validation mechanism of the monitor accepts a request if and only if the request satisfies the following two conditions:

- The page tables in *L1* and *L2* blocks do not map a block in the physical memory region allocated to Linux as both writable and executable:
    - The request specifies the modification of a page table entry in an *L1* or *L2* block such that the modified entry does not map a block as both writable and executable. A page table is in an *L1* or *L2* block if and only if τ is equal to *L1* or *L2* for the block that the page table is located in, respectively.
    - The request specifies the modification of a page table entry in an *L1* or *L2* block such that if the modified entry maps a block as writable/executable, then no other entry in a page table in an *L1* or *L2* block maps the block as executable/writable, respectively. That is, if a block is to be mapped as writable, its entry in ρ_{ex} must be equal to zero, and if a block is to be mapped as executable, its entry in ρ_{wt} must be equal to zero.

  This condition prevents Linux from writing unsigned code into blocks that are mapped as executable.
- The contents of each block that is requested to be mapped as executable has its signature in the golden image. The golden image is located in the physical memory region allocated to the monitor, and is initialized to the set of signatures that represent the code that the user of the system trusts.

When Linux reschedules a process, the hypervisor must change the set of page tables that are used by the MMU. Since the hypervisor and the monitor have already validated the page tables in blocks of type *L1* and *L2*, the hypervisor and the monitor do not need to revalidate the set of page tables that the rescheduled process uses. The hypervisor only needs to read τ for the block that contains the first-level page table of the rescheduled process. Hence, τ enables efficient switching of page tables. Also, by having the page tables in the physical memory region allocated to Linux and mapping them as writable only in privileged mode, and since Linux can configure the page tables through the memory mapping request handlers, it is not necessary to have duplicate (shadow) page tables in the hypervisor. Hence, this design uses hypervisor memory efficiently.

The memory mapping request handlers and their associated data structures are described in deeper detail in Sections 3.5 and 3.6, respectively, and more formally in Appendix B. Those two sections also describe the conditions that must be checked in order for a memory mapping request to be secure in the presence of the NIC. For instance, the memory mapping request handlers must not set blocks to be of type *L1* or *L2* or map them as executable if the NIC can write such blocks. Also, the NIC register write request handlers must not configure the NIC such that it can write blocks of type *L1* or *L2*, or that have their entry in ρ_{ex} greater than zero. Otherwise the NIC could potentially give the control of the system to Linux and enable execution of unsigned Linux code.

The design of the memory mapping request handlers has been proved in the theorem prover HOL4 to ensure that only signed Linux code is executed (on the system consisting only of an ARMv7 CPU and a memory). The verification has been done by proving this property on a HOL4 model that describes the operation of the design as a transition system. That transition system has the following characteristics:

- The state encodes the hardware state, including CPU registers and memory, some critical data structures of the hypervisor, and the golden image of the monitor.
- The execution of one CPU instruction located in Linux memory that does not cause a change in execution mode is described by one transition. The operation of such a transition is described in HOL4 by an ARMv7 ISA model [84] and an MMU model [86].
- The execution of one CPU instruction located in Linux memory that do cause a change in execution mode, potentially resulting in an invocation of a memory mapping request handler (supervisor call instruction), and the operations performed by the CPU when executing the corresponding memory mapping request handler, are described by a single transition.

The proof of that only signed Linux code is executed has been implemented in HOL4 by formulating an invariant that implies that all executable blocks have signed contents. By ensuring that all initial states satisfy the invariant and by proving that all transitions preserve it, it is proved that all reachable states satisfy the invariant. Hence, it is proved that for all system executions, all executed CPU instructions located in Linux memory are located in blocks that have signed contents. Figure 12 illustrates this proof approach, and how the transition system that represents the design differs from a transition system that represents the implementation of the binary code of the hypervisor and the monitor.

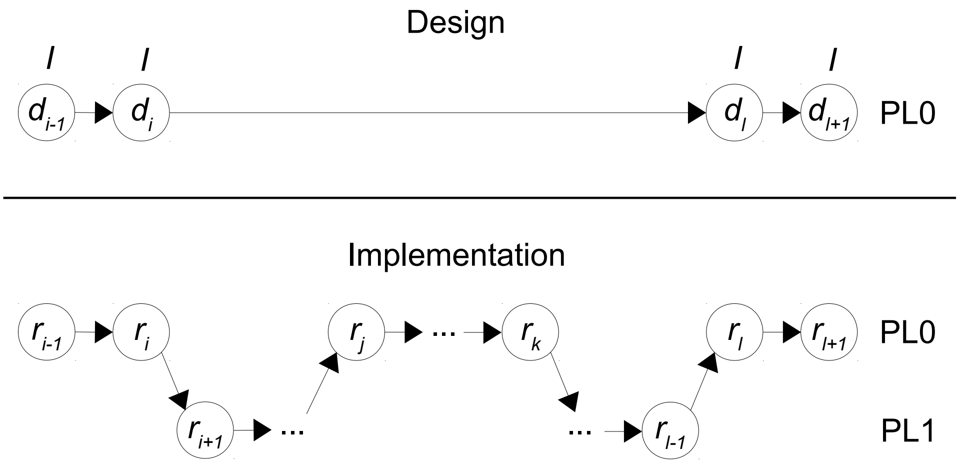


Figure 12: The difference between the transition system that describes the design and the transition system that describes an implementation of the design. The upper part of the figure shows three transitions in the former transition system. The transitions d_{i-1} → d_i and d_l → d_{l+1} describe executions of CPU instructions performed by Linux. The transition d_i → d_l describes how the design of the memory mapping request handlers handles a memory mapping request issued by Linux. This transition describes the execution of a supervisor call instruction by Linux and the operations of the invoked memory mapping request handler. The proof of that the design ensures that only signed Linux code is executed is based on that all reachable states satisfy the invariant I, and from states satisfying I, Linux blocks mapped as executable contain signed code. The lower part of the figure shows a number of transitions in the transition system that describes an implementation of the design. The transitions r_{i-1} → r_i and r_l → r_{l+1} describe executions of CPU instructions performed by Linux and manipulate the state in an identical way as done by the transitions d_{i-1} → d_i and d_l → d_{l+1}, respectively. The transition r_i → r_{i+1} describes a supervisor call exception that invokes a memory mapping request handler. The transitions from the state r_{i+1} to the state r_j and from the state r_{k+1} to the state r_l describe executions of CPU instructions of the hypervisor when its part of a memory mapping request handler is executed. The transitions from the state r_j to the state r_{k+1} describe executions of CPU instructions of the monitor when its part of a memory mapping request handler is executed. The CPU is in non-privileged mode in the states between r_{j-1} and r_{k+1} since the monitor is executed in non-privileged mode. To prove that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed, it must be proved that the transitions from the state r_i to the state r_l operate in a corresponding way as the transition d_i → d_l operates.

For the purpose of formally verifying that only signed Linux code is executed, there are two reasons for why it is suitable to use a hypervisor and a monitor. First, the hypervisor and the monitor consist of a relatively small amount of code (less than 10000 lines of code combined), compared to the Linux kernel (millions of lines of code). Second, by implementing the signed code mechanism inside a securely separated software component (the monitor), the hypervisor can retain its focus on managing the hardware and providing separation between the software components. The decoupling of the purposes of the hypervisor from the purposes of the monitor allows the verification of the separation properties of the hypervisor to be decoupled from the verification of the signed code mechanism of the monitor. The first property of code size is critical in order to make it feasible to formally verify that only signed Linux code is executed. The second property of decoupling the security mechanisms simplifies the verification compared to if the two security mechanisms were integrated in the hypervisor.

It is also interesting to know why the verification has been done in HOL4. The ambition is to formally verify that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed. The reason for such detailed verification at the ISA level is because malicious code injection attacks are often performed by means of sophisticated techniques. The verification must therefore consider the interaction between the hardware and the software. Hence, a detailed and complex hardware model of the instruction set architecture of the ARMv7 CPU must be used.

Static analyzers and tools that verify annotated source code are unsuitable for verification at the ISA level, since they are normally designed to verify program properties and not system properties that involve both hardware and software. Furthermore, verifying that only signed Linux code is executed requires analysis of all reachable hardware states in all system executions. Model checkers are unsuitable for such analyzes since they suffer from the state-space explosion problem. Since theorem provers do not suffer from these two problems, they are suitable tools for verifying that only signed Linux code is executed.

Since the implementation of complex hardware models is difficult and time consuming, and the CPU is a complex hardware component in an embedded system, it is desirable to use the already implemented model of the ARMv7 ISA in HOL4 [84]. For these reasons, HOL4 is a suitable tool to use to formally verify that only signed Linux code is executed in an embedded system containing an ARMv7 CPU with memory.

2.4 Device Model Framework and Related Theorems

The reasoning in the proof plan in Chapter 6 depends on the models described in Chapter 5, which in turn depend on the device model framework. The device model framework describes the execution of computer hardware consisting of an ARMv7 CPU, a memory and a set of arbitrary I/O devices. Basing the models on the device model framework makes the work described in this thesis usable for PROSPER. This section describes the device model framework and some theorems proved about it from the perspective of this thesis [85].

2.4.1 Device Model Framework

The device model framework is implemented in HOL4 and integrates the following HOL4 models:

- ARMv7 ISA and memory [84]: Describes the ARMv7 ISA including the operations performed on memory. This model is a transition system where

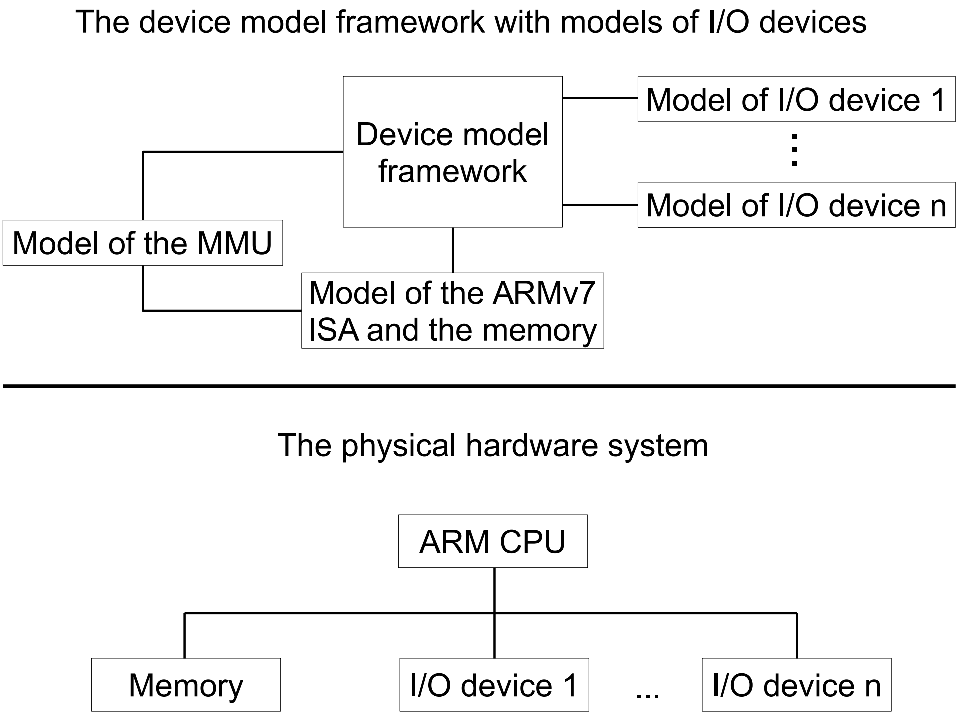


Figure 13: The device model framework integrates all HOL4 models that describe the ARMv7 ISA, the MMU, the memory and the I/O devices. The resulting model describes how the represented physical hardware system executes. The model of the ARMv7 ISA and the model of the MMU describe the execution of an ARMv7 CPU. The model of the ARMv7 ISA uses the model of the MMU to compute access permissions and physical addresses. The framework uses the output of the MMU model to decide whether the computed physical addresses correspond to memory locations or registers of I/O devices. If a physical address identifies the location of a register of an I/O device, the framework lets the model of the accessed I/O device to update its state. Interrupts from I/O device models are forwarded by the framework to the model of the ARMv7 ISA.

each transition describes the execution of one binary encoded ARMv7 instruction. The state records the contents of CPU registers and memory.

- ARMv7 ISA MMU [86]: Describes the operation of an ARMv7 ISA MMU. This model is a function, called *mmu*, that is further described in Subsection 5.2.4.3. By combining this model with the model of the ARMv7 ISA and the memory, a model of the ARMv7 CPU is obtained.
- I/O devices: Each model of an I/O device describes the operation of that I/O device as a transition system. The device models describe how the devices respond to register accesses performed by the CPU, execute independently of the CPU, access memory and raise interrupts. The framework and the I/O device models together describe when interrupts are forwarded to the CPU, and with the ARMv7 ISA and MMU models how the CPU accesses I/O device registers.

Figure 13 illustrates how the framework integrates all models.

The framework is a transition system where the state includes the state of the ARMv7 ISA and memory model and the states of all I/O device models. The transitions of the CPU and the devices are performed in an interleaved manner as determined by a non-deterministic scheduler. The scheduler determines non-deterministically whether the CPU or a device shall perform the next transition, and if it is a device, which device. The framework can therefore generate all possible interleavings of the transitions that are performed by the CPU and the devices. Figure 14 illustrates this idea, and additional ideas introduced below.

The I/O device interface of the framework consists of four types of transitions:

- Register read: The framework lets a device perform this kind of transition when the CPU reads a register of that device. Devices can therefore react and update their states when their registers are read by the CPU.
- Register write: Similar to register read transitions but with respect to register writes.
- Autonomous: A device performs an autonomous transition when it is scheduled by the non-deterministic scheduler. This transition corresponds to one autonomous execution step of a device.
- Memory read request reply: When a device performs an autonomous transition, it might generate a memory request. If it is a read request, the framework reads a byte in memory and then gives the byte value to the device. The device performs this kind of transition to react to the reply and update its state.

The rest of this subsection describes how the framework is implemented and what the interface to the device models is. This description helps the understanding of the structure of the NIC model described in Section 5.1 and Appendix C.

The transition system that the framework and the I/O device models constitute advances by means of system execution cycles. Each cycle consists of one CPU transition, followed by zero or more autonomous transitions performed by the devices, each of which is potentially followed by a memory read request reply transition. The function *next* describes one system execution cycle in the two steps (see also Figure 14). The first step lets the CPU perform a transition and access a device register, and the accessed device to perform a register read or write transition. The second step lets the devices perform an autonomous transition and potentially also a memory read request reply transition.

The two steps of *next* are implemented by the framework as follows:

1. The CPU performs one transition that corresponds to the execution of one instruction. If the instruction raises an exception, the CPU enters a state where the exception has been taken. In that case, *next* performs step 2. If no exception is raised, the instruction is executed. That might cause the CPU to access the virtual address space. If the virtual address is mapped by the MMU to a device register, *next* applies the function of the model of the accessed device that describes the register read or write transition,

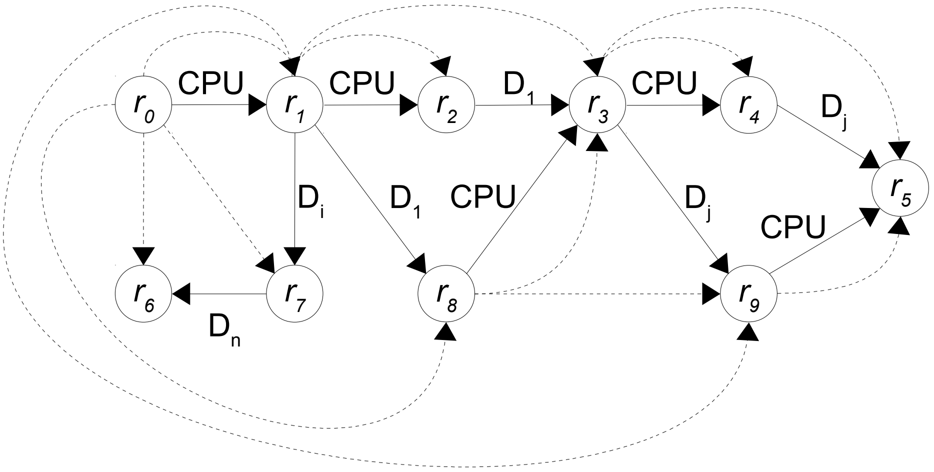


Figure 14: Several possible execution traces that can be generated by the non-deterministic scheduler in the device model framework. Transitions performed by the CPU and four devices are shown. CPU transitions have the label CPU and the device transitions have the label D indexed with 1, i, j or n, where the index identifies the device performing the transition. Some of the transitions performed by the CPU and the devices describe operations on distinct resources, and therefore can several traces end up in the same state. For instance, there are two traces from r_0 to r_3: $r_0 \rightarrow r_1 \rightarrow r_2 \rightarrow r_3$ and $r_0 \rightarrow r_1 \rightarrow r_8 \rightarrow r_3$. The former trace starts with two CPU transitions followed by a transition performed by the device D_1, and the latter trace starts with one CPU transition followed by one transition performed by the device D_1, and one additional CPU transition. One CPU transition corresponds to the execution of one CPU instruction. If the CPU instruction reads or writes a device register, the CPU transition involves two transitions (where the intermediate state is not shown): One transition of the CPU followed by one transition of the accessed device. The operation of the latter transition is described by either d_read or d_write, depending on whether the CPU instruction reads or writes a register of the accessed device. The functions d_read and d_write are instantiated by the model of the accessed device. One device transition is described by the framework function advance_single and might also involve two transitions, both of which are performed by the same device. advance_single first applies progress, which describes one execution step of the device, called autonomous transition. If the autonomous transition of the device issued a memory read request, advance_single also applies receive to let the device react on the received memory read request reply. Both progress and receive are instantiated by the model of the device performing the device transition. The framework function next describes one system execution cycle, which involves one CPU transition followed by zero or more device transitions. Transitions described by next are shown as dashed arrows without a label.

depending on whether the access is a read or a write. These two functions are referred to as *d_read* and *d_write* and have the following interface:

- *d_read*:

(device, word32) *d_read*(device *d*, word32 *pa*).

d_read takes as arguments the state of the accessed device, *d*, and the physical address, *pa*, of the device register to read. The return value is a tuple with two components. The first component is the state that the device enters when its register located at physical address *pa* is read in the state *d*. The second component is the 32-bit value of the read register when the device is in the state *d*.

- *d_write*:

  device *d_write*(device *d*, word32 *pa*, word32 *value*).

  d_write takes three arguments: the state *d* of the device whose register located at physical address *pa* is to be written with the value *value*. The return value is the state the device enters as a result of the write.

2. As long as the scheduler decides that a device shall make progress, the function *advance_single* is applied on the state of the scheduled device. Each such application of *advance_single* makes the scheduled device perform one autonomous transition. If the autonomous transition issues a memory read request, *advance_single* also lets the device perform a memory read request reply transition. Potentially raised interrupts during this step can be handled by the CPU during the next system execution cycle when *next* is applied again.

The function *advance_single* also operates in two steps:

1. *advance_single* applies the function *progress*. *progress* is instantiated by the model of the scheduled device and describes how that device performs autonomous transitions:

   (device, mem_req ∪ {⊥}, bool) *progress*(device *d*).

   The argument is the state of the device. The return value is a tuple with three components: the updated device state resulting from an autonomous transition from the device state *d*, a possible memory request, and a boolean flag that is true if the device asserts an interrupt in the updated device state *d*. A memory request is issued if and only if the second component is distinct from ⊥. The data type mem_req contains information of whether the memory request reads or writes memory, the physical address of the memory byte to access, and if it is a write, which value to write.

2. If *progress* in step 1 returned a memory request, *advance_single* applies the function *mem_acc_by_dev* on the memory request to make the request take effect. For read requests the device must be given the reply to perform a memory read request reply transition. *mem_acc_by_dev* gives the reply to the device by applying the function *receive*. *receive* is instantiated by the model of the device issuing the memory read request and describes how that device performs memory read request reply transitions:

   device *receive*(device *d*, mem_req *memory_reply*).

   receive takes as arguments the state of the device and the reply to the memory read request returned by *progress* in step 1. The return value is the updated device state that the device enters when it is given the reply in the state *d*.

The functions *d_read*, *d_write*, *progress* and *receive* are instantiated by four functions of the NIC model, *read_nic_register*, *write_nic_register*, *nic_execute* and *memory_byte*, respectively, described in Subsection 5.1.1. These latter functions describe the corresponding operations of the NIC on BeagleBone Black. Also, the proof plan in Chapter 6 is based on two models, called the real model and the ideal model, to reason that only signed Linux code is executed. The real and ideal models are described in Sections 5.2 and 5.3, respectively, both of which are based on the device model framework instantiated with these four NIC model functions. Section 5.2 describes both a part of the state and the transitions of the device model framework instantiated with the NIC model.

2.4.2 Theorems about the Device Model Framework

The theorems proved in HOL4 about the device model framework [85] not only concern the device model framework with general I/O devices. They also state properties of hypervisors with guests executed on the hardware described by the device model framework instantiated with I/O device models.

The assumptions of the theorems are the following properties and configurations of the hypervisor and the I/O devices:

- Guest memory isolation: The MMU is configured such that the execution of the current guest can only access the memory of that guest.
- Devices do not both access memory and assert interrupts: The reason behind this assumption is to provide confidentiality between guests. It prevents the guests from concluding properties about the memory contents of other guests by analyzing their own execution time.
- Between two executed instructions of a guest, devices accessing memory must either only access the memory of the current guest, or the memory of other guests: This assumption prevents devices from transferring memory contents between different guests.
- Devices do not enter undefined states: This assumption is necessary in order to analyze the execution of the devices.
- Devices do not access device registers: This assumption prevents devices from reconfiguring themselves into insecure states or to send information to other devices and guests.
- Guests cannot access device registers: This assumption prevents guests from reconfiguring devices to enter insecure states and forward information from one guest to another.

The conclusions of the theorems from these assumptions are:

- Non-infiltration: The executions of the current guest and the devices that cannot access the memory of other guests are independent of the rest of the system. This means that the execution of a guest and the devices only accessing the memory of that guest, cannot deduce any information about or be affected by the state of:

  - Other devices only accessing the memory of other guests.
  - Other guests.
  - The hypervisor.

- Non-exfiltration: The executions of the current guest cannot affect memory other than its own nor the devices that do not access the memory of that current guest. This means that the execution of a guest and the devices only accessing the memory of that guest, cannot affect the state of:
  - Devices not accessing the memory of the current guest.
  - Other guests.
  - The hypervisor.

Parts of these two conclusions would be useful for proving that only signed Linux code is executed. However, since the monitor must access hypervisor and Linux memory, and since the NIC asserts interrupts due to memory accesses, the first two assumptions about guest isolation and memory accesses versus interrupts do not hold. The theorems therefore cannot be directly applied for proving that only signed Linux code is executed. Subsection 6.6.2 discusses the applicability of these theorems with respect to the implementation of the proof plan described in Chapter 6.

2.5 Related Work

This section is structured as follows. To get some ideas of how the NIC register write request handlers can be designed and implemented, Subsection 2.5.1 describes how hypervisors developed by industry and research projects handle NICs and I/O devices. Since these handlers must be formally verified, Subsection 2.5.2 focuses on verification of device drivers. The proof plan shall not only consider the NIC but the complete system, and it is therefore relevant to know what other projects have done in the context of verifying software that controls hardware and supervises software running on top of it. Subsection 2.5.3 is therefore devoted to the verification of hypervisors and operating systems. Subsection 2.5.4 describes some tools that may be useful when implementing the proof plan presented in Chapter 6.

2.5.1 Virtualization of I/O Devices

The ARM CPU used in this project has no dedicated hardware that can be used by the hypervisor to restrict which memory regions the NIC can access. It is interesting to see which hardware assistance that exists, in case the hypervisor is ported in the future to another architecture. ARM, AMD and Intel all provide virtualization technologies for certain CPUs that can be used by hypervisors to give guests secure direct access to I/O devices. The hypervisor configures the system such that the I/O devices allocated to a guest can only access the memory of the guest in question [37-39]. From the perspective of this project and with the

NIC in mind, would such hardware probably significantly simplify the software design, the proof plan, and their implementations. The hardware assistance provided by ARM is briefly described in Section 4.4.

Several hypervisors [40-50] use one or a combination of the following three solutions to give their guests access to I/O devices:

- Emulation: The hypervisor allows the guests to access an I/O device with their original device driver. The hypervisor makes all guests believe that they have exclusive access to the device and prevents the guests from interfering with each other and the hypervisor. Emulation is commonly implemented by software in the hypervisor.
- Paravirtualization: The hypervisor have one part of a device driver, the back-end, and the guests have a paravirtualized driver, the front-end. A guest uses its front-end driver to communicate with the back-end driver, which in turn performs the actual device configuration.
- Direct access: The guests are trusted and are given direct access to the I/O devices, or the hypervisor uses virtualization support from the hardware to prevent the guests from arbitrarily configuring the I/O devices.

One interesting method for implementing secure and performance efficient NIC handling in the Xen hypervisor is by semi-automatically transforming a guest operating system device driver to a secure and efficient hypervisor device driver [51]. The hypervisor driver is created by binary rewriting the guest OS driver. The hypervisor driver only operates on guest data, which are located in guest memory. The hypervisor driver therefore cannot corrupt hypervisor data structures, which gives security. By means of an address translation mechanism, the hypervisor driver accesses guest data in guest memory without context switches. The avoidance of context switches is the key to achieving good performance. This method improved network performance for the guests by more than a factor two compared to the original Xen hypervisor.

However, this approach is not useful for the system consisting of the hypervisor, the monitor and Linux for three reasons. First, the advantage of semi-automatic generation of the hypervisor driver from the guest OS driver is not useful since the NIC register write request handlers are not concerned with managing the operation of the NIC, which the Linux NIC driver is, but only with ensuring that writes to the NIC registers are secure. Second, the advantage of the hypervisor driver not accessing hypervisor data structures is not feasible since the data structures that are used to prevent Linux from insecurely configuring the NIC must be located in the hypervisor. Third, the advantage of performance improvement due to the avoidance of context switches is not relevant since the implementation of the hypervisor allows the hypervisor and Linux to switch execution on the CPU without context switches.

2.5.2 Formal Verification of Device Drivers

The first three paragraphs are devoted to theorem proving and the last paragraph to model checking and static analysis.

Alkassar et al. [52] have verified a UART device driver. The model is a transition system that describes a system consisting of a CPU and a UART device. Each transition of the CPU corresponds to the execution of one CPU instruction, and the UART device performs transitions when it receives input from an external environment or when the CPU accesses its registers. The transitions of the CPU and the UART device are non-deterministically interleaved. The UART device cannot access memory and it interacts with the CPU by means of interrupts and its device registers. A device driver for the UART device is presented in assembly code, consisting of 11 CPU instructions. A proof is also sketched of that the assembly code terminates and that *n* words from memory are sent to the external environment. All models are formalized in the Isabelle/HOL theorem prover.

Similar work to the UART device driver verification has been done but for an ATAPI hard disk [53]. It is proved in Isabelle/HOL that an ATAPI device driver consisting of 32 CPU instructions has the following property: For all executions, after the driver has terminated, a specific page in memory has been copied to a sector of the disk. In addition, it is described how execution steps of the ATAPI device can be reordered in a system execution trace with respect to execution steps of other devices if they do not interfere, and if only the ATAPI device driver controls the ATAPI device.

Duan [54] describes a design of a general abstract device model and its integration with a HOL4 model of an ARMv7 CPU. By concluding that CPUs are faster than I/O devices, the devices are modeled as executing in terms of CPU clock cycles. This conclusion is used to justify the modeling of the execution steps of the CPU and the devices to occur in parallel instead of in an interleaved manner. The abstract device model is instantiated with a model of a UART chip that cannot access memory. A few device driver routines for this chip, including interrupt handlers that have some special properties, have been proved to be correct at the ISA level with respect to liveness and safety properties.

There are also examples where device drivers have been verified by means of model checking and static analysis tools. Functional correctness of a multi-sector read operation of a device driver for a flash device has been verified by means of several model checkers [55]. A Linux USB keyboard device driver has been verified by means of source code annotation [56]. The verified properties include absence of data races, no illegal memory accesses and correct API usage. A device driver used in an industrial critical system has been verified by means of a modified version of the static analyzer ASTRÉE [57]. The device driver is written in C and the hardware it controls is described by another C program as ghost code. (Ghost code is added to a program for the purpose of verification and does not affect the original program behavior [92].) These two C programs are composed to a multi-threaded C program to model the parallel execution of the device driver on the CPU and the controlled hardware. The multi-threaded program is given to the modified version of ASTRÉE, which proves that neither the device driver nor the controlled hardware transfers data incorrectly.

2.5.3 Formal Verification of Operating Systems and Hypervisors

The VeriSoft XT project has focused on verification of operating systems [58-60]. A microkernel operating system has been verified in Isabelle/HOL [58]. The microkernel is implemented in C0 (subset of C) with inline assembly. The verification has been achieved by modeling the system with several abstraction layers. It has then been proved the adjacent abstraction layers operate similarly, and therefore allowing a property holding on the more abstract layer to be transferred to the less abstract layer below it. By using a C0 compiler, whose correctness has been verified, this verification establishes correctness at the ISA level. Inline assembly is handled by including low-level information in the machine state, and merging a sequence of assembly instructions into an atomic operation. This allows reasoning at the C0 source code abstraction layer without the need to consider assembly instructions. I/O devices are modeled as deterministic transition systems and communicate with the CPU and an external environment in an interleaved manner. In order to simplify the proofs, the transitions of the CPU and the devices are reordered when their operations are independent.

The VeriSoft project has also verified a page fault handler of a microkernel, written in C0 with inline assembly [59]. The handler interacts with an ATAPI hard disk by means of polling. The approach is similar to the one described by Alkassar et al. [53] and uses the techniques presented by Alkassar et al. [58].

Starostin et al. [60] have made formal models and proofs of microkernel primitives. The microkernel primitives are used to implement system calls and are implemented in C0 with inline assembly. The inline assembly instructions are analyzed by means of a function that maps C0 variables to memory locations. If an assembly instruction modifies a memory location that is mapped to from a C0 variable, the value of the C0 variable is changed accordingly.

The NICTA research center has verified several properties of the seL4 microkernel. Klein et al. [61] describe how seL4 for ARMv6 and ARMv7 was designed in order to make it verifiable. The proofs for the functional correctness of the binary code of seL4 are also described and discussed. The verification has been done in Isabelle/HOL and some verified properties are correctness of optimized inter-process communication handlers, correct access control checks, no leakage of information, and for each system call its worst case execution time. Also, Klein et al. [62] describe how refinement has been used in the verification of seL4 to prove that properties that hold on more abstract models also hold on more concrete models. The formally verified C compiler CompCert [63] has also been involved in the verification with a few details described by Fernandez et al. [64].

The hypervisor Muen, which runs on multicore x86-64 CPUs with virtualization support, has been proved in Isabelle to have no runtime errors [50]. It has been implemented in a toolchain called SPARK 2014 [65, 66].

Sanán et al. [67] describe some initial verification work of the XtratuM hypervisor. An abstract security model and a model of the C code of the XtratuM hypervisor has been implemented in Isabelle/HOL. Some proof methods and concerns have been reviewed and compared to the verification of the seL4 microkernel.

The PROSPER project involves formal verification of properties of hypervisors for ARMv7 [68-71], including the hypervisor used in the work described in this thesis. An ARMv7 hypervisor has been verified at the ISA level [68]. The verification was done by means of HOL4, an ARMv7 ISA model [84], BAP [80] (static analyzer that verifies ARM assembly code), an in-house tool that transforms ARM assembly code to an intermediate language used in BAP, and the SMT solver STP (used to check if logical formulas are true). Some of these tools were used to automate the verification, and a verification example of a hypercall is described by Dam et al. [68].

A hypervisor hosting two guests has also been verified [69]. By means of the techniques presented by Dam et al. [68] it is verified that the two guests can affect each other only by means of a hypervisor provided communication channel. The proof approach is based on bisimulation between two transition system models. One model specifies how the implemented system shall behave, and another model describes the implemented system. The specification model describes a system consisting of two physically isolated ARMv7 CPUs executing in an interleaved manner. Each guest is assigned its own CPU and hypercalls are executed atomically. The implementation model describes the real system consisting of one ARMv7 CPU with the two guests running on top of the hypervisor. It is on the specification model that it has been proved that the guests can only affect each other by means of the communication path provided by the hypercalls. The bisimulation result between the two models establish that the guests make the same observations in the two models. By verifying that the binary code of the hypercalls operates as the atomic versions in the specification model, it is established that the binary code of the hypervisor is correct. The proved property on the specification model can for these two reasons be transferred to the implementation model. (This is the approach of the proof plan described in Chapter 6.)

Dam et al. [70] presents a design and a verification of a hypervisor that can run a Linux guest. The verification is done on a HOL4 model describing an ARMv7 CPU, including an MMU, and the atomic executions of a set of hypercalls of the hypervisor. The latter are used by Linux to configure its memory mapping. The model is specified as a transition system where each transition describes the execution of one CPU instruction of Linux, or the execution of one hypercall. It is then proved that executions of Linux cannot affect critical system resources or deduce any information about them (Linux is isolated). This work has been continued, where a more concrete version of the model has been added in which the hypervisor stores its data in memory instead for in abstract state components [71]. It has then been proved that these two models operate similarly, to allow the isolation property to be transferred to the concrete model. The goal is to prove this property at the ISA level.

There has also been some work with model checkers. For instance, Vasudevan et al. [49] presents the design, implementation and verification of the XMHF hypervisor. The core of the hypervisor provides security functionalities that are commonly provided by hypervisor based security architectures. It is proved that software running at a lower privilege level than the hypervisor cannot modify hypervisor memory, where DMA operations are also considered. Most of the verification is done at the C source code level with the model checker CBMC. The rest of the verification is done manually, and involves assembly code and certain parts of the C code.

The static analyzer VCC, which verifies annotated C code, has been used to prove functional correctness of a hypervisor [72, 73]. The hardware is described by means of ghost code in VCC which has then been used to prove the correctness of the C code portions of the hypervisor [72]. The assembly code has not been verified but its contracts have been specified. The correctness of the assembly code was verified at a later step by translating the assembly code to C code which was then verified by VCC [73].

PikeOS is a hypervisor that has also been verified in VCC [74]. The verification is based on models that describe the operations of the hardware and the assembly code of the hypervisor. The execution of the assembly code on the hardware model is described by C functions. These C functions are annotated and automatically verified in VCC.

2.5.4 Tools for Formal Verification

There are several tools that can be used to prove properties at the source code level, such as VCC [75], VeriFast [76], and CBMC [77]. However, in this project is verification at the ISA level desirable. Some interesting tools for verification at the ISA level are described in what follows.

The Verified Software Toolchain, VST, [78] is a toolchain that can be used to prove properties about C programs in the Coq theorem prover. VST uses the formally verified C compiler CompCert [63] to guarantee that the properties proved at the top-level also hold at the assembly level. CompCert translates a C program to an ARM assembly program by means of seven intermediate languages, and the reasoning is performed at the third most abstract language of these nine languages [79].

BAP [80] can be used to formally verify and analyze binary code for ARM. An approach to automatically verify C code with inline ARM assembly by means of model checking is described by Fehnker et al. [81].

A method for verifying x86 assembly code is presented by Maus [82] and is similar to the methods used in other work [73, 74]. Perhaps these methods can be used in the context of ARM assembly as well. The operations performed by the CPU are described by ghost code. The x86 assembly code is verified by translating it to C code by means of the tool Vx86, and the translated C code is then annotated and verified automatically in VCC.

Li [83] describes a compiler for a functional language, TFL, that is a subset of the HOL4 language. The compiler is implemented and proved correct in HOL4 and targets ARM. The user can specify algorithms in TFL, prove properties about them in HOL4, and then compile the algorithms to ARM instructions. Since the compiler is proved to be correct, the proved properties of the algorithms also hold on the ARM instructions. It is also described how C programs that only use a small subset of C can be imported into TFL.

3 Software Design

This chapter describes the software design of the memory mapping request handlers [86], their extensions that consider the operation of the NIC, and the NIC register write request handlers. The purposes of these handlers are to enable Linux to configure the MMU and the page tables, and the NIC, respectively, but only under certain conditions. Namely, these handlers must ensure that the hypervisor, the monitor and Linux are securely separated, and that all executable blocks contain signed code in the physical memory region allocated to Linux. Otherwise, the control of the system is potentially given to Linux or the CPU might execute unsigned Linux code.

3.1 Without the Network Interface Controller

To ensure that the software components are securely separated (referred to as the separation property) and that the content of all executable blocks is signed (referred to as the execution property), these memory regions must be protected. In the original system without the NIC, all memory accesses go through the MMU, which allows or blocks the memory accesses according to the configuration of the page tables. Hence, the MMU and the page tables must be protected as well. Since only the memory mapping request handlers have access to the MMU and the page tables, these handlers are responsible for establishing the separation and execution properties. The separation property is established by the hypervisor, and the execution property by the monitor.

To be more precise, the resources in the original system that must be protected by the memory mapping request handlers are:

- Hypervisor memory: Contains the code, heap and stack of the hypervisor, including the parts of the memory mapping request handlers that ensure the separation property, and the critical data structures τ, ρ_{wt} and ρ_{ex}. The memory mapping request handlers depend on τ, ρ_{wt} and ρ_{ex} to preserve the separation and execution properties. Other critical code that is also stored in this memory region is, for instance, the code that returns the CPU to Linux in non-privileged mode. This memory region must be protected from both Linux and the monitor.

  If Linux can read this memory region, Linux could potentially read confidential data. If Linux can write into this memory region, Linux can change the code of the memory mapping request handlers, and the contents of the data structures τ, ρ_{wt} and ρ_{ex}. Such writes can therefore compromise or circumvent the checks of the memory mapping request handlers to enable Linux to execute unsigned code. These writes can also overwrite code in the hypervisor such that the CPU is returned to Linux in privileged mode, giving Linux complete control of the system.

  The monitor is restricted to only being able to read ρ_{wt} and ρ_{ex}. This restriction eases the verification of the separation and execution properties, since the verification of these two properties can then be performed independently.

- Monitor memory: Contains the part of the code of the memory mapping request handlers that ensures the execution property, and the critical golden image. This memory region must be protected from Linux. If Linux can read this memory region, Linux might get access to confidential data. If Linux can write into this memory region, Linux can modify the code of the memory mapping request handlers or the golden image, enabling Linux to execute arbitrary code.
- Linux memory:
  - Executable blocks: Contain code that Linux can execute. These blocks must be write-protected from Linux and have signed content. Otherwise Linux can write and execute unsigned code.
  - Page tables: Contain data determining access permissions of all blocks in the physical address space. The page tables must have secure configurations and the physical memory blocks containing them must be write-protected from Linux and the monitor. Otherwise, Linux can write blocks allocated to the hypervisor or the monitor or storing executable code. The monitor should not have access to page tables since they are a hardware resource maintained by the hypervisor.
- The TTBR0 register: Contains the physical address of the first-level page table used by the MMU to start its translation table walks. It must be protected from Linux and the monitor, and point to a block of type *L1* to make the MMU use page tables with secure configurations. Otherwise, Linux can potentially take control of the system or execute unsigned code. The monitor shall not have access to TTBR0 for the same reasons as for the page tables.
- The DACR register: Contains the two-bit codes used by the MMU to determine how to compute access permissions. Must be protected from Linux and the monitor for the same reasons as for the TTBR0 register.

Figure 15 shows where these five resources are located and what they contain in the system.

3.2 Threats from the Network Interface Controller

The NIC shall be exclusively used by Linux (not shared with the hypervisor or the monitor) such that the NIC device driver in Linux can configure the NIC as it is programmed to, provided that the configurations are compatible with the separation and execution properties. Normally, the NIC device driver in Linux configures the NIC such that the NIC does not violate these two properties. However, if the NIC device driver in the Linux kernel or the Linux itself for some reason deviate from their ordinary behavior (e.g. due to bugs, or malicious software, whose signatures are incorrectly a part of the golden image), they might attempt to configure the NIC such that the separation or execution property does not hold. Such deviating NIC configurations could enable the NIC to write

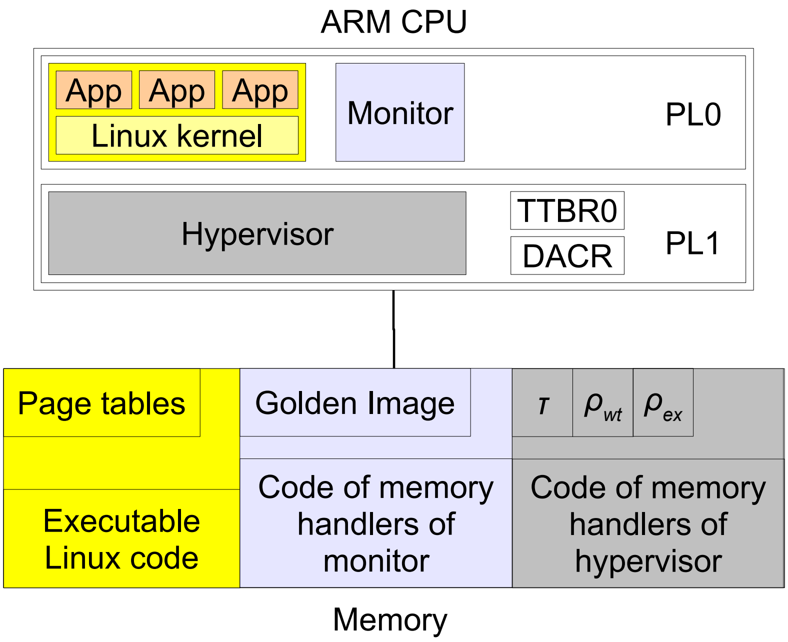


Figure 15: The original software design [86] in the system without the NIC. The critical resources that must be protected by the hypervisor are the TTBR0 and DACR registers, the blocks containing page tables and executable code in the memory region allocated to Linux, and the memory regions allocated to the monitor and the hypervisor which contain the golden image, τ, ρ_{wt}, ρ_{ex}, the code of the memory mapping request handlers and critical exception handling code of the hypervisor. Linux is executed in non-privileged mode and cannot access TTBR0 nor DACR since these registers are only accessible in privileged mode. Linux can only access its own memory region, but not write blocks containing page tables or executable code. The monitor is executed in non-privileged mode and cannot access TTBR0 nor DACR. The monitor can only access its own memory region, read the blocks containing executable code in the memory region allocated to Linux, and read the block in the hypervisor that contains ρ_{wt} and ρ_{ex}. The hypervisor is executed in privileged mode, can access all resources in the system, and maintains τ, ρ_{wt} and ρ_{ex}. The code of the memory mapping request handlers executed by the monitor and the hypervisor is located in the memory region allocated to the monitor and the hypervisor, respectively.

received frames in any of the four memory regions that the separation and execution properties depend on: hypervisor and monitor memory, and blocks in Linux memory that are executable or store page tables. Hence, the NIC registers that affect which memory accesses the NIC performs must also be protected from Linux and the monitor, along with the resources described in the list in the previous section. Since the NIC cannot access TTBR0 and DACR, both of which are CPU registers, it is sufficient to protect these NIC registers that affect which memory accesses the NIC performs, in order to preserve the separation and execution properties when the system is extended with the NIC.

Since reads of NIC registers have no side effects that affect which memory accesses the NIC performs, it is enough to write-protect the eight registers included in Figure 9 in Subsection 2.3.2 and the two NIC registers RX_BUFFER_OFFSET and DMACONTROL. The operation of the NIC with respect to the eight NIC registers included in Figure 9 is described in Subsection 2.3.2, and the latter two registers are described in Section C.1. It is writes to these ten NIC registers that the NIC register write request handlers must check to not violate the separation and execution properties. Furthermore, the data structures that the NIC register write request handlers depend on must also be write-protected from Linux and the monitor. Since the hypervisor manages the hardware and the NIC register write request handlers manage the NIC, these handlers are a part of the hypervisor.

The memory mapping request handlers must be extended to prevent blocks that are writable by the NIC to be allocated to store page tables or mapped as executable.

3.3 Overview of the Software Design

To enable the memory mapping request handlers to be aware of which blocks that are writable by the NIC, the data structure ρ_{NIC} is introduced. ρ_{NIC} takes a block index as argument and returns the number of buffer descriptors in the receive queue of the NIC that address the block with the given index. Since the NIC only writes blocks that are addressed by buffer descriptors in the receive queue, it is sufficient for the memory mapping request handlers to test if ρ_{NIC} is equal to zero for block indexes whose corresponding block is to be allocated to store a page table (the block is to be typed by τ as *L1* or *L2*) or mapped as executable (ρ_{ex} is to be incremented for the block). If the entry in ρ_{NIC} for the given block index is greater than zero, it means that there is at least one buffer descriptor in the receive queue of the NIC that address the block with the given index. Hence, the NIC can write the block with that index, and therefore the memory mapping request handlers must reject requests that specify such blocks to be allocated to store a page table or mapped as executable. When a buffer descriptor is added to the receive queue, ρ_{NIC} is incremented by one for the block index of each block that the buffer descriptor addresses, and similarly decremented when the buffer descriptor is released by the NIC.

To write-protect the ten NIC registers affecting which memory accesses the NIC performs from Linux and the monitor, those registers are mapped by the page tables as read-only in non-privileged mode. If Linux attempts to write those registers, a data abort exception occurs and causes the data abort exception handler of the hypervisor to execute. The data abort exception handler reads the DFAR register (described Subsection 2.3.1) to find out which virtual address Linux attempted to write. If the virtual address is mapped to a physical address of a NIC register, the data abort exception handler reads the link register to identify the virtual address of the instruction whose execution caused the exception. The data abort exception handler hands these two addresses to a top-level function of the NIC handling code.

From the two given virtual addresses, the top-level function of the NIC handling code identifies which NIC register Linux attempted to write and the instruction whose execution caused the exception. By parsing that instruction, the value that

Linux attempted to write can be computed. The top-level function then gives that value to the NIC register write request handler that checks writes to the identified NIC register.

Each NIC register write request handler has three tasks:

- Preserve the separation property: Ensuring that the NIC is configured such that it does not access blocks allocated to the hypervisor or the monitor and does not write blocks containing page tables.

  This property is preserved by restricting which blocks the transmission and reception queues of the NIC are allowed to address. For the transmission queue, the buffer descriptors are only allowed to address blocks that are allocated to Linux. The NIC therefore cannot read confidential data located in the memory allocated to the hypervisor or the monitor. For the reception queue, the buffer descriptors are only allowed to address blocks allocated to Linux and which do not contain page tables. The NIC therefore cannot write blocks allocated to the hypervisor or the monitor or storing page tables. The buffer descriptors in the transmission queue are allowed to address the page tables since that does not break the separation property and it avoids unnecessary restrictions on Linux, although Linux normally does not transmit page tables. Hence, the NIC cannot access blocks allocated to the hypervisor or the monitor and not write blocks containing page tables.

- Preserve the execution property: Ensuring that the NIC is configured such that it does not write blocks allocated to the hypervisor or the monitor, or containing page tables or executable code. (The contents of all of these blocks are relevant for establishing the execution property for the following reasons. Blocks allocated to the hypervisor and the monitor contain the code of the memory mapping request handlers that establish the execution property, or the NIC register write request handlers that ensure secure configuration of the NIC. The part of the memory mapping request handlers that is located in the hypervisor and is related to the execution property is the invocation of the monitor and the handling of the response from the monitor. The blocks containing page tables are used to restrict Linux from writing into executable blocks, while the executable blocks contain code that must be signed. These are the reasons why blocks of these four types are necessary to protect.)

  The execution property is preserved by means of the restrictions described in the previous bullet item (buffer descriptors are only allowed to address Linux memory and not write page tables), and by not allowing any buffer descriptor in the receive queue to address an executable block. All buffer descriptors in the receive queue are allowed to only address blocks whose type is *D* and entries in ρ_{ex} are equal to zero. Hence, the NIC cannot write blocks allocated to the hypervisor or the monitor, or containing page tables or executable code.

- Preserve the NIC in a defined state: The NIC must always be in a defined state. If the NIC enters an undefined state, its operation is unknown and the

> NIC may then potentially perform operations that break the separation or execution property. Section 5.1 and Appendix C describe a model of the NIC that formally describes when the NIC enters an undefined state. That model is used by the proof plan in Chapter 6 to formally verify that NIC register write request handlers do not configure the NIC to enter an undefined state. The NIC register write request handlers must therefore prevent configurations of the NIC that cause the NIC to enter an undefined state as described by the NIC model.
>
> To ensure that the NIC is always in a defined state, several boolean flag variables are used. These flags allow the handlers to know which operations the NIC is currently performing. By reading these flags, the handlers can avoid configuring the NIC to initiate additional operations while the NIC is performing its current operations, which would otherwise cause the NIC to enter an undefined state.

If the invoked NIC register write request handler determines that the write Linux attempted to perform is compatible with the purposes of the three tasks described above, the handler re-executes the write to make it take effect. Otherwise, the handler does not perform the write and informs the data abort exception handler of this violation. If the write is not performed, it means Linux that deviated from its normal behavior. How this situation should be handled depends on how the system is used. For instance, the data abort exception handler can return control to Linux, reboot Linux, or print an error message and freeze the system.

In the general case when a data abort exception occurs and the virtual address in the DFAR register is not mapped to a physical address of a NIC register, it means that the data abort exception is not related to a NIC register write. Also, if a data abort exception occurs when a Linux application executes, it means that an application is attempting to perform privileged operations, since only the Linux kernel (specifically the NIC device driver inside the Linux kernel) shall have access to the NIC registers. In both of these two cases, if the data abort exception is not related to the restriction of the monitor that requires blocks to not be both writable and executable, the data abort exception is not relevant to the hypervisor and the exception is forwarded to the data abort exception handler in the Linux kernel (see the descriptions of prefetch and data abort exceptions in Subsection 2.3.4.1).

There are two additional data structures that deserve special attention in this design overview section. *tx0_active_queue* and *rx0_active_queue* contain the physical addresses of the heads of the transmission and reception queues (for DMA channels zero which are the only ones Linux uses), respectively. These two queues contain all buffer descriptors that are currently in use by the NIC (see Figure 10 in Subsection 2.3.2.2). Hence, these two data structures enable the NIC register write request handlers to know which parts of CPPI_RAM that are in use by the NIC. This information is critical, since the contents of CPPI_RAM affect which memory accesses the NIC performs. The data structures mentioned in this section and more thereto are described in detail in the next section.

Figure 16 shows all critical resources in the software design and in the original system extended with the NIC. The three parts of the software design are:

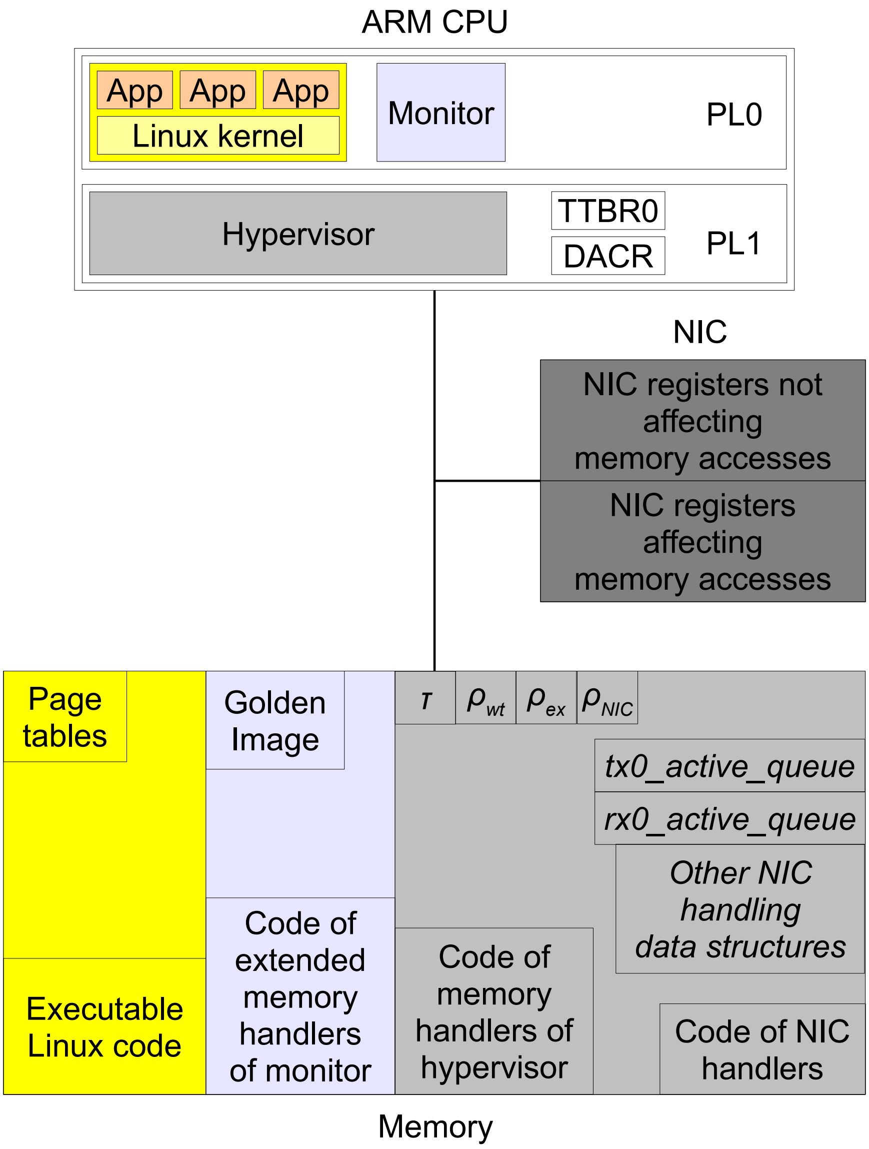

ARM CPU
App
App
App
Linux kernel
Monitor
PL0
Hypervisor
TTBR0
DACR
PL1
NIC
NIC registers not affecting memory accesses
NIC registers affecting memory accesses
Page tables
Golden Image
τ
ρwt
ρex
ρNIC
tx0_active_queue
rx0_active_queue
Other NIC handling data structures
Executable Linux code
Code of extended memory handlers of monitor
Code of memory handlers of hypervisor
Code of NIC handlers
Memory

Figure 16: The software design in the original system extended with the NIC. Additional critical resources compared to Figure 15 that must be protected by the hypervisor are the NIC registers that affect which memory accesses the NIC performs (see Figure 9 in Subsection 2.3.2), the data structures the NIC register write request handlers maintains (shown as ρ_{NIC}, tx0_active_queue, rx0_active_queue and "Other NIC handling data structures"), the code of the extended part of the memory mapping request handlers, and the code of the NIC register write request handlers. These data structures and the NIC register write request handlers are located in the memory region allocated to the hypervisor. Only the hypervisor has access to all of these additional resources, except for ρ_{NIC} which can also be read by the monitor. Since only the monitor's part of the memory mapping request handlers accesses ρ_{NIC} (which only reads ρ_{NIC} to prevent blocks writable by the NIC to be allocated to store page tables or mapped as executable), only the monitor's part of the memory mapping request handlers needs to be extended. Also, only the hypervisor executes the NIC register write request handlers, which update ρ_{NIC}.

- The data structures that are used by the extended memory mapping and NIC register write request handlers.
- The extended memory mapping request handlers (referred only to as the memory mapping request handlers for the rest of this thesis).
- The NIC register write request handlers.

These three parts are described in the following three sections.

3.4 Data Structures

The data structures that the memory mapping and NIC register write request handlers use are:

- bool *initialized*

  Set to true when the NIC DMA hardware has been reset and the HDP and CP registers have been initialized to zero. This means that the NIC has been initialized. It is set to false when the least significant bit of the CPDMA_SOFT_RESET register is written to one to initiate the reset operation of the NIC DMA hardware. This variable is used to prevent the NIC from entering an undefined state, which occurs if certain NIC registers are written to initiate certain NIC operations when the NIC has not been initialized.

- bool *tx0_hdp_initialized*, *rx0_hdp_initialized*, *tx0_cp_initialized*, *rx0_cp_initialized*

  These variables are set to true when the corresponding HDP or CP register has been zeroed after the NIC DMA hardware reset operation has completed. When all these variables have been set to true, it is known that all HDP and CP registers have been initialized, at which point *initialized* is set to true. These variables are set to false when the least significant bit of the CPDMA_SOFT_RESET register is set to one. The only purpose of these variables is to inform the NIC register write request handlers of when *initialized* shall be set to true.

- bool *tx0_tearingdown*, *rx0_tearingdown*

  These variables are set to true when the teardown operation of transmission or reception DMA channel zero is initiated, respectively. That occurs when TX_TEARDOWN or RX_TEARDOWN is set to (channel) zero. These variables are set to false when the corresponding teardown interrupt is acknowledged. A teardown interrupt is acknowledged when the corresponding CP register contains 0xFFFFFFFC and it is written with the same value. At that point, it is known that the teardown operation has completed.

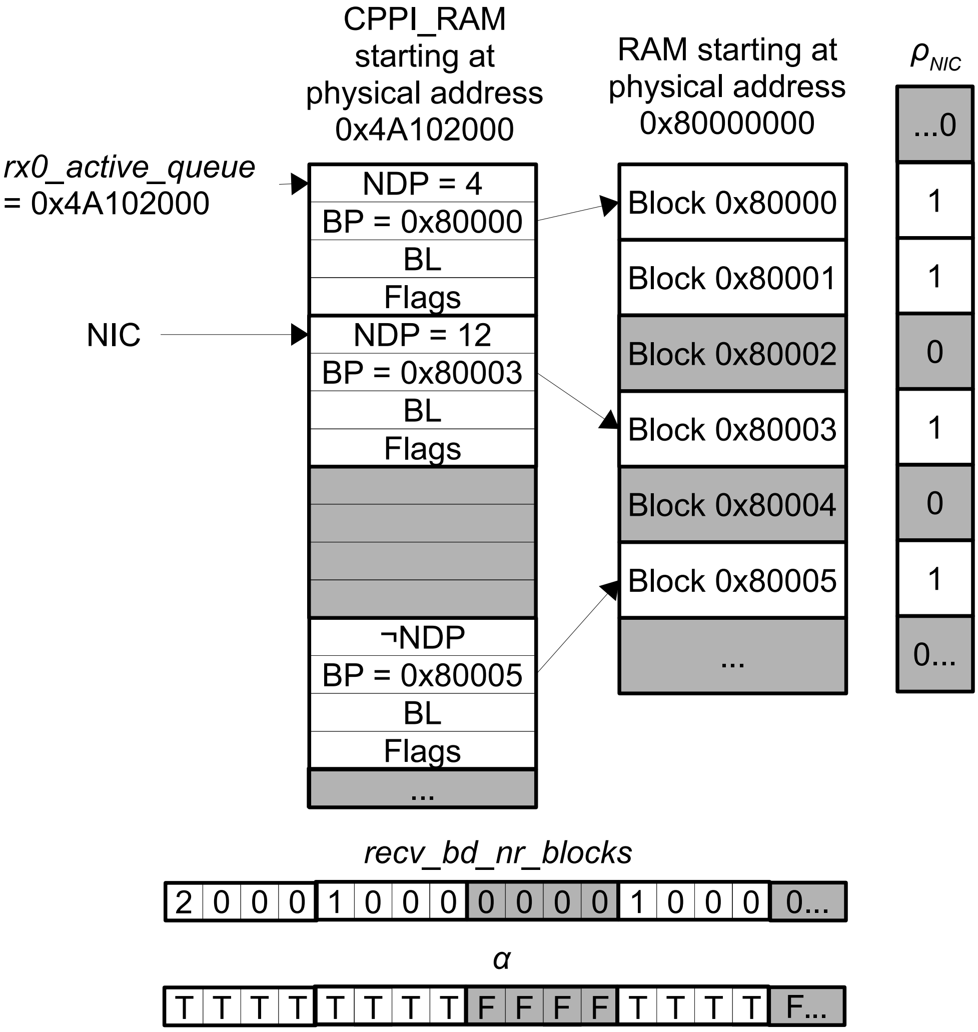


Figure 17: An example of the status of the data structures rx0_active_queue, ρ_{NIC}, recv_bd_nr_blocks and α, when the NIC is in a state where it only processes received frames. The NIC has processed one buffer descriptor and has two left that it can use to store received frames. Shaded areas in CPPI_RAM and RAM are not accessed by the NIC, and their corresponding entries in ρ_{NIC}, recv_bd_nr_block and α are also shaded. Since each memory block is addressed by at most one buffer descriptor in the queue pointed to by rx0_active_queue, their entries in ρ_{NIC} are either zero or one. NDP is an abbreviation for next descriptor pointer, BP for buffer pointer and BL for buffer length. The Flags field corresponds to the fourth word of a buffer descriptor and is not important in this example. NDP is set to the 32-bit word index of CPPI_RAM for where the next buffer descriptor in the queue is located, and ¬NDP means a next descriptor pointer value equal to zero. BP is set to the block index of the block containing the start of the data buffer that the buffer descriptor addresses. In this figure, NDP and BP use indexes of words in CPPI_RAM and blocks in RAM instead of addresses for simplification. For recv_bd_nr_blocks and α, only the first 16 entries are shown, since the rest are set to zero and false, respectively. The entries in recv_bd_nr_blocks indicate that the first buffer descriptor in the queue identified by rx0_active_queue addresses two blocks while the following two buffer descriptors only address one block.

These variables are used to prevent the NIC from entering an undefined state. If certain NIC registers are written to initiate certain NIC operations while the NIC is performing a teardown operation, the NIC can enter an undefined state. In order to prevent such writes, *tx0_tearingdown* and *rx0_tearingdown* are used to record the state of the NIC with respect to teardown operations.

- word32 *tx0_active_queue*, *rx0_active_queue*

  Contain 32-bit physical addresses of buffer descriptors that are heads of buffer descriptor queues used for transmission and reception, respectively. These two queues contain all buffer descriptors that are currently in use by the NIC for transmission and reception, respectively. The purpose of these two variables is to inform the NIC register write request handlers how CPPI_RAM is currently used by the NIC. When an HDP register is written with a physical address of a buffer descriptor to initiate transmission or enable reception, the corresponding variable is set to the same physical address. These two variables are then updated by certain NIC register write request handlers to bypass buffer descriptors in the queues that have been released by the NIC. When the NIC has processed a complete queue, the corresponding variable is set to zero. Figure 17 shows an example that illustrates the role of *rx0_active_queue*. Other data structures shown in Figure 17 are described by the following bullet items.

- {*L1*, *L2*, *D*, *MN*, *N*, ⊥} *τ*(word20 *bl*)

  τ is a function that takes as argument a block index, *bl*, of 20 bits, to the physical address space, and returns the type of the block with the given block index *bl*, identified by an element in the set {*L1*, *L2*, *D*, *MN*, *N*, ⊥}. These symbols have the following meaning:

  - *L1*, *L2*: The block is a physical memory block that contains a first- or second-level page table that has been verified by the memory mapping request handlers. The memory mapping request handlers can therefore set TTBR0 to point to a block of type *L1* without performing any security checks.
  - *D*: The block is a physical memory block that contains arbitrary Linux data.
  - *MN*: The block corresponds to a 4 kB physical address region that contains NIC registers that affect which memory accesses the NIC performs. Such block indexes are in {0x4A100, 0x4A102, 0x4A103}.
  - *N*: The block corresponds to a 4 kB physical address region that contain NIC registers but none of which affect which memory accesses the NIC performs. Its index is equal to 0x4A101.
  - ⊥: The block is outside the memory region allocated to Linux, or the block is not mapped by any page table in an *L1* or *L2* block.

  See Figure 18 for an example of the status of *τ*. Blocks of type *L1*, *L2* or *D* belong to the memory region allocated to Linux, while blocks of type ⊥

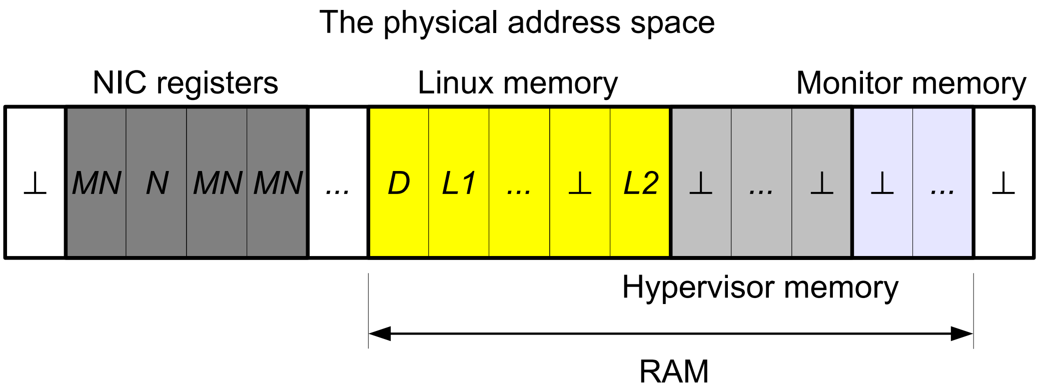


Figure 18: An example of the status of τ. τ is constant for all block indexes that correspond to blocks that are outside Linux memory. The NIC registers constitute four blocks, where three contain registers that affect which memory accesses the NIC performs. Of the blocks allocated to Linux, one is shown as being unmapped by the symbol ⊥.

are not accessible to Linux. τ does not only allow secure and efficient switching of page tables when modifying TTBR0, but also efficient security checks of several memory mapping and NIC register write requests. For instance, if a new receive buffer descriptor only addresses blocks not storing page tables. (τ is equal to D for such blocks. Such blocks could also potentially be of type $\perp$, but buffer descriptors normally only address blocks that are mapped by the page tables. Hence, $\perp$ is not relevant.)

- word32 ρ_{wt}(word20 *bl*) , ρ_{ex}(word20 *bl*), ρ_{NIC}(word20 *bl*)

  For a given block index *bl*, these functions record the following data (represented as 32-bit strings):

  - ρ_{wt} and ρ_{ex} record the number of page table entries in *L1* and *L2* blocks that map the block with index *bl* as writable and executable for Linux, respectively.
  - ρ_{NIC} records the number of buffer descriptors in the queue pointed to by *rx0_active_queue* that address the block with index *bl*. (See Figure 17.)

  These functions simplify the operation of the memory mapping and NIC register write request handlers. ρ_{wt} and ρ_{ex} are read by the monitor to check that page table configurations do not map blocks as both writable and executable to Linux. ρ_{ex} is read by the NIC register write request handlers to check that buffer descriptors in the receive queue do not address blocks that are executable by Linux. ρ_{NIC} is used by the memory mapping request handlers to check that blocks writable by the NIC are not allocated to store page tables (to be typed as *L1* or *L2*; checked by the hypervisor) nor mapped as executable (checked by the monitor).

  In fact, ρ_{wt} and ρ_{NIC} can be integrated into a single function to avoid modifications of the original memory mapping request handlers. Both ρ_{wt}

and ρ_{NIC} record the number of write references of a block, and it does not matter for the separation and execution properties whether Linux writes a block through the CPU or the NIC. The two entries in ρ_{wt} and ρ_{NIC} for each block index can therefore be added with the result stored by a single function. However, ρ_{wt} and ρ_{NIC} are not integrated in this thesis for clarity.

- word32 *recv_bd_nr_blocks*(word11 *i*)

  The argument *i* is an 11-bit index of a 32-bit word in CPPI_RAM (which consists of 8 kB). It returns the number of blocks addressed by the buffer descriptor whose first 32-bit word has the index *i* in CPPI_RAM. For instance, if the first word of a buffer descriptor starts at physical address 0x4A102000, (first word of CPPI_RAM), the index is zero, and if the first word starts at 0x4A102004, the index is one. (See Figure 17.) The entries in *recv_bd_nr_blocks* are updated when a buffer descriptor is added to the queue pointed to by *rx0_active_queue* and when *rx0_active_queue* is updated to bypass a buffer descriptor.

  This function is used to correctly update ρ_{NIC} when *rx0_active_queue* is updated to bypass released buffer descriptors. When a receive buffer descriptor is given to the NIC, that buffer descriptor is added to the tail of the queue pointed to by *rx0_active_queue*. At that point, ρ_{NIC} is also updated for the blocks that the added buffer descriptor addresses by reading the buffer pointer and buffer length fields of that buffer descriptor. The issue is that if a received frame does not fill the data buffer addressed by the buffer descriptor, the buffer length field is changed by the NIC to the actual amount of data stored in the data buffer. If the frame is small enough, the buffer pointer and buffer length fields can indicate that only one block is addressed by the buffer descriptor instead of two. Using the buffer length field to update ρ_{NIC} when *rx0_active_queue* is updated to bypass released buffer descriptors could therefore result in that ρ_{NIC} have incorrect values. *recv_bd_nr_blocks* is used to solve this problem.

- bool α(word11 *i*)

  The argument is an index to a 32-bit word in CPPI_RAM (as for *recv_bd_nr_blocks*) and the return value is true if and only if the identified 32-bit word is a part of any buffer descriptor in the queues pointed to by *tx0_active_queue* and *rx0_active_queue*. (See Figure 17.) If Linux attempts to write a buffer descriptor that is in use by the NIC, then that write might affect which memory accesses the NIC will perform. Hence, such writes must be checked to not break the separation and execution properties. By reading α, the NIC register write request handlers can easily and efficiently check whether Linux attempts to write a buffer descriptor that is in use by the NIC. If Linux attempts to write a buffer descriptor that is not in use by the NIC, no checks are necessary and the write can be re-executed. α is updated when buffer descriptors are added to the queues pointed to by *tx0_active_queue* and *rx0_active_queue* and when these two variables are updated to bypass released buffer descriptors.

- <wordx> *GI*

The golden image is a set of bit strings of length *x*. Each bit string is a signature of code of size 4 kB (size of a block). Each signature in the golden image represents code that is trusted by the user of the system. *x* shall be replaced by the actual bit string length of a signature.

3.5 Memory Mapping Request Handlers

The memory mapping request handlers are hypercalls invoked by Linux when Linux allocates, deallocates and modifies its page tables, and switches the first-level page table identified by TTBR0. They ensure that configurations of page tables respect the separation and execution properties, and that TTBR0 points to a validated page table. This section describes these original handlers [86], and their extensions needed in the presence of the NIC. The extensions consist of additional requirements that the memory mapping request handlers must check to be satisfied in order for the execution of a memory mapping request to not break the separation and execution properties. The memory mapping request handlers are described more formally in Section B.2.

The handlers operate on the data type ideal_state that is used in Section 5.3 to define a model that in turn is used in the proof plan in Chapter 6. Instances of this data type contain the state of the hardware and the state of the data structures described in the previous section. The second component of the returned tuple is a flag that is true if and only if the handler executed the given request.

The memory mapping request handlers are the following functions:

- (ideal_state, bool) *switch*(ideal_state *i*, word20 *bl*)

  Switches the first-level page table that the MMU uses to the one contained in the block with index *bl*. This function checks that $\tau(bl) = L1$, meaning that the new page table has been validated.

- (ideal_state, bool) *freeL1*(ideal_state *i*, *word20 bl*),
  (ideal_state, bool) *freeL2*(ideal_state *i*, *word20 bl*)

  Changes the type of the block with index *bl* from *L1*/*L2* to *D*. Linux and the NIC are then allowed to write that block if it is not executable ($\rho_{ex}(bl) = 0$). If $\tau(bl) = L1$, then TTBR0 must not point to that block. Otherwise, the MMU could potentially use a first-level page table that Linux and the NIC can write. If $\tau(bl) = L2$, then must no second-level page table link entry (called ”Page table” in Figure 8 in Subsection 2.3.1) in a page table in any *L1* block refer to the block with index *bl*. Otherwise, the MMU could potentially use a second-level page table that Linux and the NIC can write.

- (ideal_state, bool) *unmapL1*(ideal_state *i*, word20 *bl*, word10 *e*),
  (ideal_state, bool) *unmapL2*(ideal_state *i*, word20 *bl*, word10 *e*)

  Frees the page table entry with index *e* in the page table in the *L1*/*L2* block with index *bl*. That block must not be executable ($\rho_{ex}(bl) = 0$). Otherwise, this function could potentially modify the block such that it contains unsigned code, which would then be executable by Linux.

- (ideal_state, bool) *linkL1*(ideal_state *i*, word20 *bl*, word10 *e*, word20 *bl'*)

Sets the page table entry with index e in the page table in the $L1$ block with index bl to point to the second-level page table in the $L2$ block with index bl'. That is, the first-level page table in the former block gets a link to the second-level page table in the latter block. For this operation to be executed, $\rho_{ex}(bl) = 0 \wedge \tau(bl') = L2$ must hold. The former conjunct ensures that this operation does not make an executable block contain unsigned code. The block with index bl' must be of type $L2$ to ensure that it has been validated. Otherwise, the MMU could potentially use an unvalidated second-level page table, which might not respect the separation and execution properties.

- (ideal_state, bool) *mapL1*(ideal_state *i*, word20 *bl*, word10 *e*, word20 *bl'*, bool *rd*, bool *wt*, bool *ex*),
  (ideal_state, bool) *mapL2*(ideal_state *i*, word20 *bl*, word10 *e*, word20 *bl'*, bool *rd*, bool *wt*, bool *ex*)

  Sets the page table entry with index e in the page table in the $L1/L2$ block with index bl to point to the block with index bl' with read, write and execute access permissions as indicated by *rd*, *wt* and *ex*, respectively. Such an entry is called ”Section”/”Small page” in Figure 8 in Subsection 2.3.1. If $\tau(bl) = L1$, then 256 consecutive blocks are mapped starting at the location of the block with index bl', all with the same access permissions. For each block that gets mapped as writable or executable, its entry in ρ_{wt} or ρ_{ex} is incremented by one, respectively. In what follows, bl'' denotes the index of a mapped block (including bl').

  For a request to be executed, it must satisfy the following requirements:

  - The block is of type $L1/L2$: $\tau(bl) = L1/L2$. This requirement prevents the handlers from wrongly believing that the block with index bl is storing a page table when it actually represents NIC registers or is allocated to the hypervisor or the monitor. Hence, this requirement prevents the handlers from writing NIC registers or memory blocks allocated to the hypervisor or the monitor. Such writes might otherwise make the NIC enter an insecure state, or change code or data structures of the hypervisor or the monitor.
  - The block with index bl is not executable: $\rho_{ex}(bl) = 0$. This requirement ensures that these two handlers do not modify executable blocks such that they contain unsigned code.
  - Blocks mapped as executable have signed contents:

    $$ex \Rightarrow sign(content(i.memory, bl'')) \in GI.$$

    content is a function that takes as arguments the state of the memory in the ideal state *i*, containing information of memory contents, and a block index bl''. *content* then returns the bit string stored in the 4 kB block with index bl'' in the ideal state *i*. *content* is formally defined in Section 6.1. *sign* is the function used by the monitor to compute signatures of block contents to determine whether the corresponding code is signed or not. (*sign* is described in subsection 5.3.3.) This requirement ensures that blocks becoming executable contain signed code.

  - Blocks must belong to the physical memory region allocated to Linux or to NIC registers:

    $$bl'' \in LINUX\_BL \lor \tau(bl'') \in \{MN, N\}.$$

    LINUX_BL contains the block indexes of all physical memory blocks allocated to Linux. This requirement prevents Linux from accessing memory allocated to the hypervisor or the monitor.

  - Executable blocks must not be writable:

    $$ex \Rightarrow \neg wt \land \rho_{wt}(bl'') = 0.$$

    Prevents Linux from writing and then executing unsigned code.

  - Writable blocks must not be executable nor used to store page tables:

    $$wt \Rightarrow \neg ex \land \rho_{ex}(bl'') = 0 \land \tau(bl'') \notin \{L1, L2\}.$$

    Prevents Linux from writing and executing unsigned code and changing access permissions in page tables to take control of the system.

  - Blocks corresponding to NIC registers affecting which memory accesses the NIC performs must not be writable:

    $$wt \Rightarrow \tau(bl'') \neq MN.$$

    Prevents Linux from configuring the NIC to enter an insecure state.

  - Executable blocks must not be written by the NIC:

    $$ex \Rightarrow \rho_{NIC}(bl'') = 0.$$

    Prevents the NIC from writing unsigned code.

  - NIC registers must not be executable:

    $$ex \Rightarrow \tau(bl'') \notin \{MN, N\}.$$

    This requirement prevents the CPU from interpreting the contents of NIC registers as executable instructions. If the CPU interprets the contents of NIC registers as instructions and executes them, Linux could potentially execute unsigned code, since the NIC can change the contents of those registers.

- (ideal_state, bool) *createL1*(ideal_state *i*, word20 *bl*),
  (ideal_state, bool) *createL2*(ideal_state *i*, word20 *bl*)

  Sets the block with index *bl* to be of type *L1*/*L2*, to enable the page table in it to be used by the MMU. The physical blocks that get mapped by the page table entries in the page table in the block with index *bl* get their entries in ρ_{wt} and ρ_{ex} updated accordingly.

  The block with index *bl* must satisfy the following requirements:

  - Each page table entry in the page table in the block must be checked as an individual entry is checked by *mapL1*/*mapL2* (see the requirements of *mapL1*/*mapL2* that mention mapped blocks denoted by *bl''*). This requirement ensures that an individual entry of a new page table that is mapping a memory block is secure with respect to the separation and execution properties.

- If the type of the block is to be set as *L1*, all second-level page table link entries in the page table in the block must point to *L2* blocks. This requirement ensures that if the page table in the block is used as a first-level page table by the MMU, the MMU will only use validated page tables.
- No entry in the page table in the block maps a block as writable if another entry in that page table maps the same block as executable, and vice versa. This requirement ensures that a single page table does not map a block as both writable and executable. (*mapL1*/*mapL2* only check that an individual entry does not map a block as both writable and executable.)
- The block belongs to the physical memory region allocated to Linux and is currently not mapped as writable: $bl \in LINUX\_BL \land \rho_{wt}(bl) = 0$; and the page table in the block does not map itself as writable. Hence, Linux cannot write newly allocated page tables. Also, the design requires page tables to be located in memory allocated to Linux, which is where Linux normally allocates page tables.
- The block must not be writable by the NIC: $\rho_{NIC}(bl) = 0$. This requirement prevents allocation of blocks writable by the NIC to be used to store page tables.

Since the block is required to be a part of Linux memory (second last requirement), it is not necessary to check that the block does not represent NIC registers. If the contents of NIC registers were interpreted by the MMU as page table entries, the NIC could potentially modify page table entries to break the separation and execution properties, since the NIC can modify the contents of its registers.

3.6 NIC Register Write Request Handlers

The NIC register write request handlers check writes to the NIC registers affecting which memory accesses the NIC performs. The handlers are invoked by the data abort exception handler of the hypervisor when Linux attempts to write such a NIC register, all of which are write-protected by the page tables. If an attempted write respects the separation and execution properties and does not configure the NIC to enter an undefined state, the hypervisor re-executes the write, and otherwise not. The NIC register write request handlers are completely specified in pseudocode, with the most important parts included in Sections B.3 and B.4.1.

The argument and return values of the NIC register write request handlers have the following meaning. The data type ideal_state has the same role as in the case with the memory mapping request handlers. Each handler as an argument called *val*, *bd_ptr* or *channel* and which is the value that Linux attempted to write to a NIC register. One handler handles writes to several NIC registers and another one handles writes to CPPI_RAM. These two handlers have an additional argument, *pa*, which is the physical address of the NIC register or location in CPPI_RAM Linux attempted to write. The return flag is true if and only if the handler re-executed the write.

The NIC register write request handlers are the following functions:

- (ideal_state, bool) *cpdma_soft_reset_handler*(ideal_state *i*, word32 *val*)

  This handler checks that Linux does not write the CPDMA_SOFT_RESET register such that the NIC enters an undefined state, according to the NIC model described in Section 5.1 and Appendix C. For this purpose, this handler reads the flags *initialized*, *tx0_tearingdown* and *rx0_tearingdown*. If CPDMA_SOFT_RESET is written to initiate the reset operation of the NIC DMA hardware, all five initialization variables are set to false.

- (ideal_state, bool) *cppi_ram_handler*(ideal_state *i*, word32 *pa*, word32 *val*)

  When Linux attempts to write CPPI_RAM, this handler checks that the write is of one of the following two types:

  - Linux attempts to write a 32-bit word in CPPI_RAM that is not a part of any buffer descriptor that is in use by the NIC: Linux performs these writes when it is initializing buffer descriptors that shall later be given to the NIC. Since the word is not in use by the NIC, these writes are not a security threat. This type of write is checked by reading α.

  - Linux attempts to write the 32-bit word that is the next descriptor pointer field of the last buffer descriptor in the queue pointed to by *tx0_active_queue* or *rx0_active_queue*: Linux performs these writes when it extends the transmission or reception queues.

    The new buffer descriptors must only address physical memory allocated to Linux, and for buffer descriptors to be added to the receive queue, they must only address non-executable blocks of type *D*. Hence, the NIC can only access Linux memory but not write page tables nor executable blocks. Also, buffer descriptors in use by the NIC are not allowed to overlap each other to prevent the NIC from rewriting their contents. The NIC writes certain buffer descriptor fields after frames have been transmitted and received (see description of NIC management fields in Subsection 2.3.2.2). If two buffer descriptors overlap, writes to certain fields of the first buffer descriptor can potentially change certain fields of the second buffer descriptor. Hence, such writes can potentially change which blocks the second buffer descriptor addresses. The NIC might then perform memory accesses that break the separation or execution properties, or perform operations that cause it to enter an undefined state. This issue is also discussed in Subsection 5.1.3.2.

    These two checks of accessed memory and non-overlapping of buffer descriptors are performed by reading α, τ and the buffer descriptors in the queues pointed to by *tx0_active_queue*, *rx0_active_queue* and *val* (which points to the queue with the new buffer descriptors). If the write is re-executed, α is updated, and if the new buffer descriptors are added to the receive queue, ρ_{NIC} and *recv_bd_nr_blocks* are also updated.

It is desirable to make the handlers as simple as possible since they are intended to be formally verified, which therefore simplifies the verification task. To make this handler as simple as possible, this handler only allows these two types of writes. Since Linux does not perform any other types of writes to CPPI_RAM, this restriction is not a practical problem.

- (ideal_state, bool) *tx0_hdp_handler*(ideal_state *i*, word32 *bd_ptr*), (ideal_state, bool) *rx0_hdp_handler*(ideal_state *i*, word32 *bd_ptr*)

  These two handlers check the two types of writes Linux performs to the TX0_HDP and RX0_HDP registers:

  - Linux attempts to initialize the NIC: After the NIC DMA hardware has been reset, the HDP registers should be initialized to zero. These two handlers check that these two registers are correctly initialized without causing the NIC to enter an undefined state. The checks are satisfied when *initialized* is equal to false, the least significant bit of the CPDMA_SOFT_RESET register is equal to zero (cleared by the NIC when it has completed the reset operation), and *bd_ptr* is equal to zero (the value Linux attempts to write). At that point, it is known that the NIC DMA hardware has been reset and that Linux attempts to reset an HDP register. Hence, these handlers write zero to TX0_HDP or RX0_HDP, and set *tx0_hdp_initialized* or *rx0_hdp_initialized* to true, respectively. If the four initialization flags of the HDP and CP registers are all true, *initialized* is also set to true.
  - Linux attempts to initiate transmission or enable reception: When Linux writes these two registers to initiate transmission or enable reception, it must be checked that the NIC has been initialized and that all buffer descriptors in the queue pointed to by *bd_ptr* are secure. The NIC is checked to be initialized by checking that *initialized* is true. All buffer descriptors are checked to be secure by performing the same checks as performed in *cppi_ram_handler* when Linux extends a transmission or reception queue.

- (ideal_state, bool) *tx0_cp_handler*(ideal_state *i*, word32 *val*), (ideal_state, bool) *rx0_cp_handler*(ideal_state *i*, word32 *val*)

  These two handlers check the three types of writes Linux performs to the TX0_CP and RX0_CP registers:

  - Linux attempts to initialize the NIC: TX0_CP and RX0_CP shall also be written to zero after a NIC DMA hardware reset operation. This check is nearly identical to the corresponding one performed by *tx0_hdp_handler* and *rx0_hdp_handler*, but with the roles of the HDP registers and the HDP initialization flags replaced by the CP registers and the CP initialization flags.

  - Linux attempts to acknowledge a teardown interrupt: The purpose of this check is to record when a teardown operation is complete in order to falsify the corresponding teardown operation flag, *tx0_tearingdown* or *rx0_tearingdown*. If the corresponding flag is true, the corresponding CP register contains 0xFFFFFFFC, and *val* is equal to 0xFFFFFFFC, it means that the corresponding teardown operation is complete and that Linux attempts to acknowledge the corresponding teardown interrupt. In such a case, the corresponding teardown operation flag is falsified.
  - Linux attempts to perform another write: If Linux does not attempt to perform any of the other two types of writes to the CP registers, these two handlers check that the write does not cause the NIC to enter an undefined state. Rejected writes are the ones attempted when the NIC performs a NIC DMA hardware reset or teardown operation, or having incorrect initialization or teardown interrupt acknowledgement values.

- (ideal_state, bool) *tx_teardown_handler*(ideal_state *i*, word32 *channel*), (ideal_state, bool) *rx_teardown_handler*(ideal_state *i*, word32 *channel*)

  These two handlers check that writes to the TX_TEARDOWN and RX_TEARDOWN registers do not cause the NIC to enter an undefined state. A write is performed to these registers if and only if the NIC has been initialized, the teardown operation to initiate is currently not in progress, and the DMA channel Linux attempts to tear down has the index zero (the NIC model only describes the operations of the DMA channels zero since Linux only uses those two channels; that is, teardowns of DMA channels with other indexes cause the NIC to enter an undefined state according to the NIC model). These check are performed by checking that *initialized* is true, the corresponding teardown operation flag, *tx0_tearingdown* or *rx0_tearingdown*, is false, and *channel* is zero. If the write is performed, the corresponding teardown operation flag is also set to true, to record that the NIC is now performing the corresponding teardown operation.

- (ideal_state, bool) *dmacontrol_handler*(ideal_state *i*, word32 *val*), (ideal_state, bool) *rx_buffer_offset_handler*(ideal_state *i*, word32 *val*)

  These two handlers check that Linux does not modify the DMACONTROL and RX_BUFFER_OFFSET registers. The purpose is to prevent the NIC from entering an undefined state. Since Linux does not write these registers, this restriction is not a practical problem.

- (ideal_state, bool) *stateram_handler*(ideal_state *i*, word32 *val*)

  This handler pretends to initialize the HDP and CP registers for all other 14 NIC DMA channels (having indexes one to seven for transmission and reception). Since those channels are unused by Linux, except when Linux initializes the NIC, they are not described by the NIC model. Writes to these registers are therefore blocked to prevent the NIC from entering an undefined state. The return flag is still set to true, pretending the writes were performed.

- (ideal_state, bool) *write_nic_register_handler*(ideal_state *i*, word32 *pa*, word32 *val*)

  This handler unconditionally re-executes writes to NIC registers Linux attempts to write and which do not affect which memory accesses the NIC performs. This handler is invoked when Linux attempts to write a register whose physical address belongs to the same block as the physical address of a register that affects which memory accesses the NIC performs. Since registers in the same block have the same access permissions, writes to the former type of register cause unnecessary data abort exceptions. This handler therefore just re-executes such writes.

3.7 Conclusion and Discussion

There are two important characteristics of this design. First, the design allows a practical implementation. The usage of data abort exceptions to invoke the NIC register write request handlers makes these handlers invisible to the NIC device driver in Linux. Hence, the NIC device driver does not need to be modified. Also, the implementation of all NIC handling code is decoupled from the rest of the hypervisor. Only the data abort exception handler of the original hypervisor must be extended. It must be extended with:

- Two CPU register reads to retrieve the necessary information: The DFAR register must be read to identify which virtual address the CPU attempted to access when the data abort exception occurred, in order to determine if a NIC register was accessed, and if so, which NIC register. Also, the link register must be read to identify the instruction whose execution caused the data abort exception, in order to determine which value the CPU attempted to write (to a NIC register).
- Two checks to determine whether the NIC handling code must be invoked: If the accessed virtual address causing the data abort exception is mapped to a NIC register and the Linux kernel is currently being executed (as opposed to a Linux application), then the NIC handling code shall be invoked and otherwise not.
- One function invocation of the NIC handling function.
- Possibly code checking and handling rejected NIC register writes.

Second, the data buffers that the buffer descriptor addresses does not need to be copied from Linux memory to hypervisor memory. The reason is that the operation of the NIC is independent of memory contents, and therefore can Linux write anything in the data buffers without affecting the separation and execution properties. However, as Table 1 in Section 4.3 shows and as discussed in Section 4.4, this design incurs some overhead that probably can be reduced by paravirtualizing the NIC device driver in Linux.

Since the separation and execution properties are normally not violated by Linux, these two properties do not prevent Linux from performing its ordinary operations. Hence, Linux is just as functional in a system with the hypervisor and the monitor as without them, excluding possible performance impacts from the hypervisor and the monitor.

Subsection 2.5.1 mentions three hypervisor designs for giving guests access to I/O devices. The design presented in this chapter for giving Linux access to the NIC is emulation, since the original NIC device driver in Linux is unmodified and interference with the hypervisor and the monitor is prevented by hypervisor software. In general, a hypervisor might be designed to provide support for allowing several guests to share the same I/O device. This is not needed for the design presented in this chapter since only Linux is given access to the NIC. Hence, the NIC register write request handlers need only ensure that the NIC is configured such that it respects the separation property (establishing the execution property is in general not a task of a hypervisor). The NIC register write request handlers are therefore relatively simple compared to the case of allowing several guests to share the NIC. However, there are also hypervisors, as described in Subsection 2.5.1, that do not support sharing nor separation (that is, their guests are trusted and given direct access to the I/O devices without hardware ensuring the I/O device accesses are secure). Section 4.4 describes how the hypervisor design could change in the case when the hardware implements the ARMv7 virtualization extensions or a system memory management unit. In such cases the execution time and network performance would be improved, and the system memory management unit would make the NIC register write request handlers unnecessary.

Some hypervisor designs also have a large trusted code base, hence affecting the reliability of the separation. For instance, the Xen hypervisor have a guest operating system, often Linux, that handles the configuration of the I/O devices [93]. The other ordinary guests communicate with this device driver guest when they need to access an I/O device. Hence, the trusted code base in such a system is the code in the Xen hypervisor plus the code in the Linux kernel. It is the Linux kernel that the design presented in this chapter attempts to secure (and the applications running on top of it)!

4 Implementation

This chapter describes the modifications of the hypervisor and the paravirtualized Linux 3.10 kernel that were made to implement the NIC register write request handlers in the hypervisor and to enable Internet access in Linux when Linux is executed on top of the hypervisor. Section 4.1 describes how the NIC register write request handlers are implemented in the hypervisor. Section 4.2 describes how the hypervisor and Linux were modified to enable Internet access in Linux. Section 4.3 presents network performance results. Section 4.4 discusses how the network performance can be improved.

4.1 Management of the NIC in the Hypervisor

The implementation in the hypervisor for giving Linux secure NIC access follows the design of the NIC register write request handlers described in Section 3.6. Since the design specifies the NIC register write request handlers to be invoked by the data abort exception handler when the execution of Linux attempts to write the NIC registers, no modifications of the Linux kernel were necessary with respect to NIC register accesses. The NIC registers are located in the physical address range [0x4A100000, 0x4A104000) which constitutes four blocks each of 4 kB. The address range of the first and the last two blocks addresses NIC registers affecting which memory accesses the NIC performs. Those three blocks are therefore mapped with read-only access permission to the CPU in non-privileged mode. Such a mapping causes the CPU to take a data abort exception if the CPU executes Linux and attempts to write a NIC register affecting which memory accesses the NIC performs. The NIC registers addressed by the address range of the second block do not affect which memory accesses the NIC performs. That second block is therefore mapped with read-write access permission to the CPU in non-privileged mode. The CPU can thus write those NIC registers when it executes Linux.

The Linux kernel as configured in this project has a region in its virtual address space at [0x80000000, 0xFF000000) that is allocated for static memory mappings (a virtual address in that region is always mapped to the same physical address). The NIC registers are mapped from such a static virtual memory region to ease the implementation. To prevent the virtual memory region mapping the NIC registers to overlap virtual memory regions mapping registers of other I/O devices, such as the timer and the interrupt controller, the NIC registers are mapped from the virtual address range [0xFA400000, 0xFA404000), as illustrated in Figure 19. This mapping is configured during the execution of the boot code of the hypervisor. The execution of that boot code also initializes the NIC by resetting the NIC DMA hardware followed by zeroing all 16 HDP and CP registers, making the NIC enter a defined idle state.

4.2 Internet Access in Linux

The following modifications were done to the paravirtualized Linux 3.10 kernel to provide its applications with Internet access:

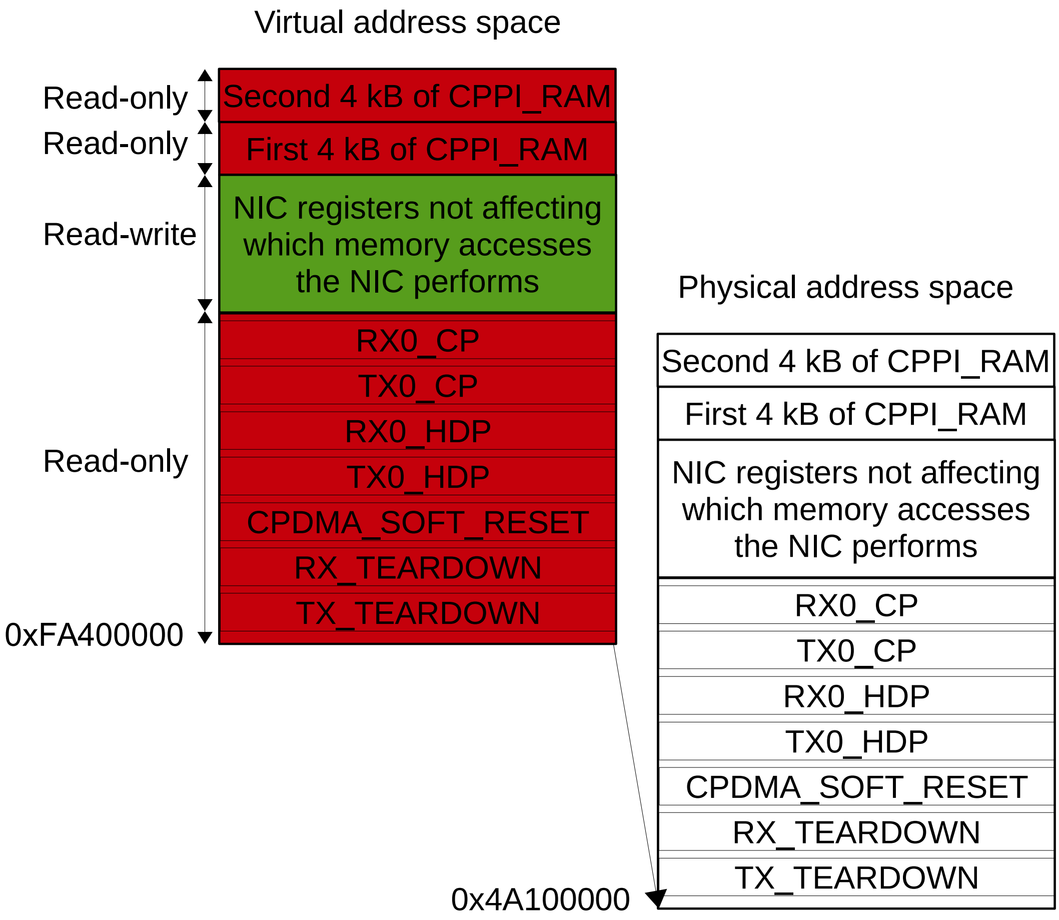


Figure 19: The virtual to physical address mapping of the NIC registers in the implementation. In non-privileged mode, in which Linux is executed, the address range of the three blocks addressing NIC registers, which affect memory accesses, is mapped as read-only. The address range of the block addressing only NIC registers not affecting memory accesses is mapped as readable and writable.

- Inclusion of the Linux networking code in the executable Linux image: This was done by extending the configuration file specifying which features of the Linux kernel that shall be included in the compilation. Only the necessary networking code is included.
- Insertion of three hypercall invocations in the Linux source code: When the CPU executes Linux in non-privileged mode, the added networking code causes the CPU to take exceptions due to attempts of performing privileged operations related to cache management and branch prediction. Hypercall invocations of hypervisor routines were therefore inserted in Linux to make the CPU perform those privileged operations.
- Specification of which virtual addresses the NIC device driver in Linux shall use to access the NIC registers: A list storing hardware related data structures was extended with one entry to make the NIC device driver in

Linux use the virtual address range [0xFA400000, 0xFA404000) to access the NIC registers.

The following modifications were done to the hypervisor to give Linux Internet access:

- Implementation of the two hypercalls related to the cache management: For simplicity, these two hypercalls were implemented in C by following the cache management assembly code in Linux. Worth to note is that the invocation of one of these two hypercalls results in deletion of cache line entries. That hypercall must not delete cache lines holding data belonging to the memory regions allocated to the hypervisor or the monitor. Otherwise hypervisor and monitor data structures can be corrupted. (The hypercall related to branch prediction was already implemented.)
- Forwarding NIC interrupts to Linux: In order to forward NIC interrupts to Linux, the boot code of the hypervisor was extended to enable the four interrupt lines assigned to the NIC. The boot code was also extended to specify that when a NIC interrupt occurs, the hypervisor shall forward that NIC interrupt to the irq exception handler of Linux.

4.3 Network Performance

The network performance for Linux applications when Linux is executed on top of the hypervisor was measured with the tool netperf [94]. BBB was connected to a PC server with a point-to-point link of 100 Mbit/s, where netperf 2.7.0 was used in Linux on BBB and netperf 2.6.0 in the PC server. The following benchmarks were performed:

- TCP_STREAM: Transfers data with TCP from netperf in Linux on BBB to netperf in the PC server.
- TCP_MAERTS: Transfers data with TCP from netperf in the PC server to netperf in Linux on BBB.
- UDP_STREAM: Transfers data with UDP from netperf in Linux on BBB to netperf in the PC server.
- TCP_RR: TCP is used to send a request from netperf in Linux on BBB to netperf in the PC server which responds to the request. Such a transaction can be considered as a ping test between two Linux applications involving no processing time at the end nodes.
- UDP_RR: As TCP_RR but with UDP.

Each benchmark was run in four different system configurations:

- The original non-paravirtualized Linux 3.10 kernel is executed natively by the CPU without involving the hypervisor or the monitor: Benchmarks run in this system configuration measure the network performance for Linux applications in an ordinary system configuration.
- Paravirtualized Linux executed on top of the hypervisor and alongside the monitor with read-write access to the NIC registers: This means that the

CPU when executing Linux can write any NIC register without causing an exception and a related security check. Benchmarks run in this system configuration measure the network performance for Linux applications by including the execution overhead of the hypervisor and the monitor, except the execution overhead caused by exceptions and executions of security checks related to NIC register writes.

- Paravirtualized Linux executed on top of the hypervisor and alongside the monitor with read-only access to the NIC registers affecting memory accesses, but where the execution of the data abort exception handler of the hypervisor immediately performs the NIC register write without any security checks: Benchmarks run in this system configuration measure the network performance for Linux applications when the execution overhead is included of the exceptions caused by writes to the address range of the three blocks addressing NIC registers affecting memory accesses.
- Paravirtualized Linux executed in the system configuration implementing the software design: Benchmarks run in this system configuration measure the network performance for Linux applications when the total execution overhead of the hypervisor and the monitor is included.

For each of the four system configurations, each benchmark was run successively five times. The results are collected in Figures 20 through 23, where the lowest value (worst result) was selected from the five benchmark tests. (For each set of five benchmark tests, all results were fairly close to each other. The selected value is therefore reasonably representative.) Notable is that the network performance is similar in all virtualized system configurations (denoted RW, ROX and V). When the message size (amount of data passed from netperf to the Linux kernel in a single send system call) is between 32 bytes and 512 bytes in the data transferring benchmarks (TCP_STREAM, TCP_MAERTS and UDP_STREAM), the throughput ratio for Linux applications is between 0.5% and 10% when Linux is executed in a virtualized system configuration compared to when Linux is executed natively. For message sizes between 1024 bytes and 16384 bytes, the throughput ratio is between 7% and 42%. For the application ping benchmarks (TCP_RR and UDP_RR), the transaction ratio is between 19% and 24%.

4.4 Conclusion and Discussion

This chapter demonstrates that the NIC register write request handlers work in practice, and that the network performance provided to Linux applications is limited. In many applications of embedded systems, where fairly small amounts of data are transferred, the network performance is probably sufficient. For mobile devices this solution is of limited use, and for multimedia applications and most data communications equipment this solution is probably of no use.

Since the network performance is similar in all virtualized system configurations, the exceptions and security checks related to NIC register writes have a small impact on network performance. Hence, the first network performance limitation seems to be the execution of Linux in non-privileged mode causing exceptions and overhead execution of the hypervisor and the monitor when the execution of Linux

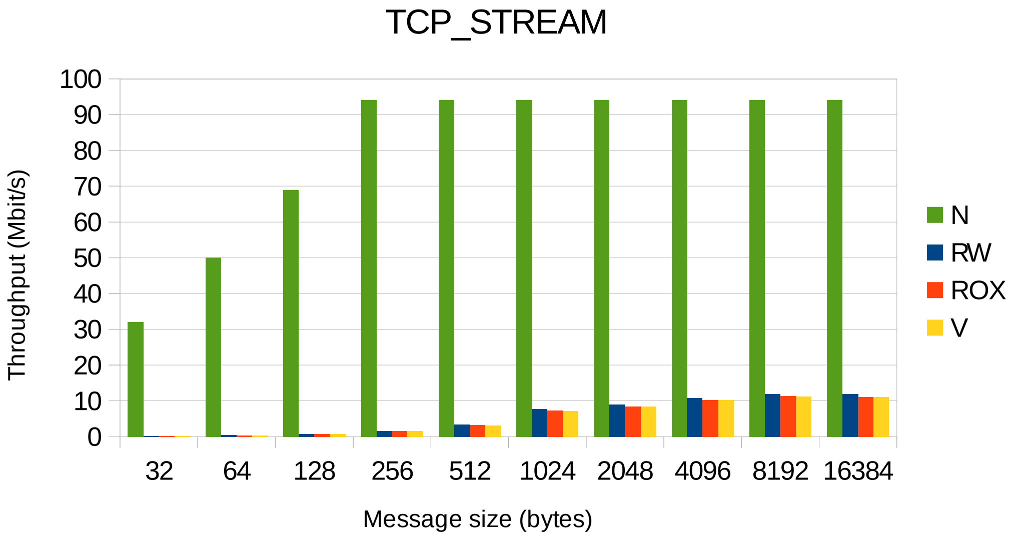


Figure 20: Results from the TCP_STREAM benchmark tests. For message sizes increasing from 32 bytes to 512 bytes, the throughput ratio for Linux applications increases from 0.5% to 4% when Linux is executed in the virtualized system configurations (RW, ROX, V) compared to when Linux is executed in the native system configuration (N). For message sizes increasing from 1024 bytes to 16384 bytes, the throughput ratio increases from 7% to 12%.

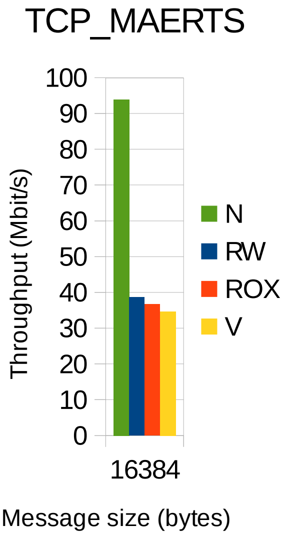


Figure 21: Results from the TCP_MAERTS benchmark tests. The throughput ratio is between 36% and 42% for Linux applications in the virtualized system configurations compared to Linux applications in the native system configuration. (No TCP_MAERTS benchmark tests were performed for other message sizes since netperf does not support varying message sizes for the TCP_MAERTS benchmark.)

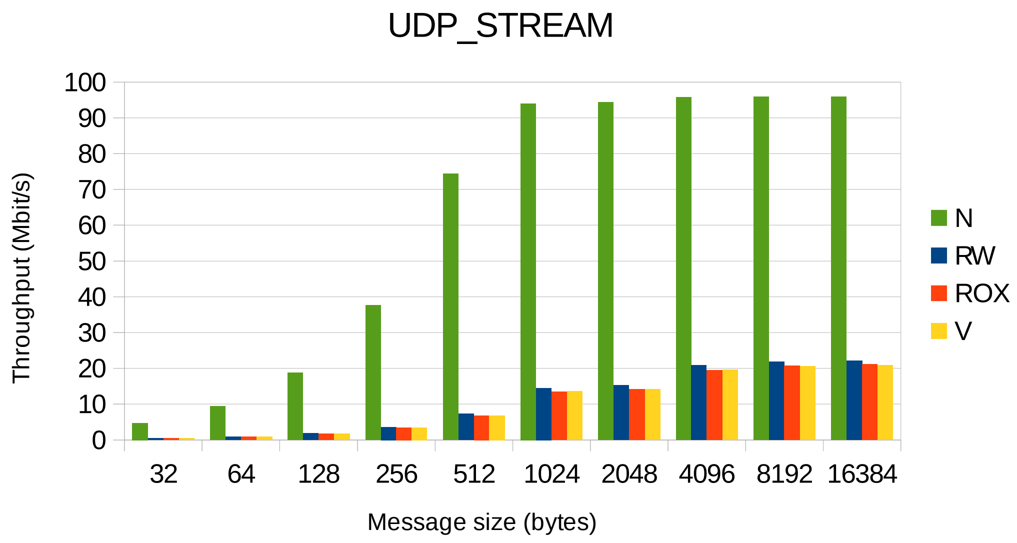


Figure 22: Results from the UDP_STREAM benchmark tests. For message sizes between 32 bytes and 512 bytes, the throughput ratio is approximately 9% for Linux applications in the virtualized system configurations compared to Linux applications in the native system configuration. For message sizes of 1024 bytes or 2048 bytes the throughput ratio is approximately 14%, and for message sizes between 4096 bytes and 16384 bytes the throughput ratio is approximately 21%.

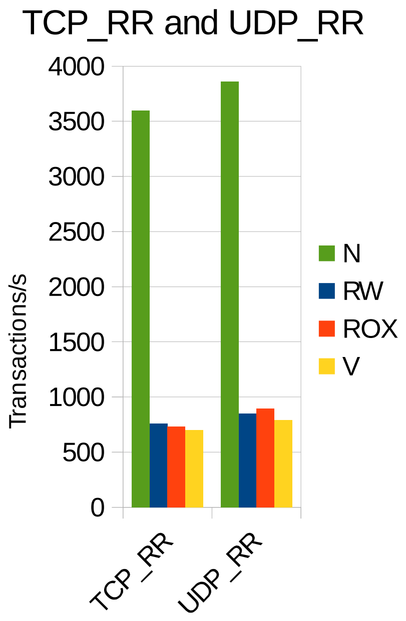


Figure 23: Results from the TCP_RR and UDP_RR benchmark tests. For TCP_RR, the transaction ratio is between 19% and 22% for Linux applications in the virtualized system configurations compared to Linux applications in the native system configuration. For the UDP_RR benchmark, the transaction ratio is between 20% and 24%.

needs to perform privileged operations (e.g. configuring the memory mapping of Linux via the memory mapping request handlers).

ARM specifies the optional virtualization extensions (VE) to the ARMv7 architecture [33] (which are not implemented in BBB). VE includes an additional privilege level and a second stage of address translation. Code executed in the additional privilege level has complete control of the system and is used to execute a hypervisor. A hypervisor can then isolate itself and the guest systems from each other by configuring the page tables of the second address translation stage such that the guest systems can only access their own memory regions and not configure I/O devices. This allows operating systems to be executed on top of a hypervisor without being modified, saving significant implementation efforts.

Raho et al. [95] presents benchmark results for guest systems executed on top of Xen and KVM (a hypervisor based on the Linux kernel) and for Linux executed natively. The used hardware is ARMv7 with VE. The execution time for the guest systems on top of Xen or KVM is at most 5% longer than the execution time of Linux executed natively. For message sizes of 64 bytes or greater, the throughput for all systems reached the maximum level. For the TCP_RR and UDP_RR benchmarks, the transaction ratio is between 67.5% and 76%.

If VE is used, the overhead from hypervisor invocations is probably reduced, but to which extent is unclear with current facts. The hypervisor and the monitor must still be invoked when a Linux block is to be mapped as executable. The reason is that the monitor must check that the signature of the content of the block is in the golden image and that the NIC cannot write the block. If that is the case the hypervisor must ensure that the page tables of the second address translation stage map the block as executable. Similarly, when a Linux block is to be mapped as writable, the hypervisor must be invoked to ensure that the page tables of the second address translation stage do not map the block as executable. Hence, the second address translation stage seems to be of limited use in this context, of ensuring execution of only signed Linux code by means of a hypervisor and a monitor. However, the additional privilege level of VE might provide functionality that make unnecessary the hypervisor invocations related to cache management, branch predication, interrupts, and other primitive privileged operations such as reading the TTBR0 register. If these operations are performed often, VE might be used to significantly improve the execution time.

Even if VE enables significant improvements in execution time, the network performance might not be improved due to the exceptions and security checks of NIC register writes. Some of this overhead can be reduced. To transmit or process a received frame, the NIC device driver in Linux attempts to perform nine NIC register writes. In addition to a few other operations, each such write causes the following operations:

- A CPU exception involving 17 writes to memory (or cache) for saving the contents of the 16 user mode registers and CPSR.
- Execution of a NIC register write request handler. Apart from the actual execution overhead of the security checks, this execution might cause

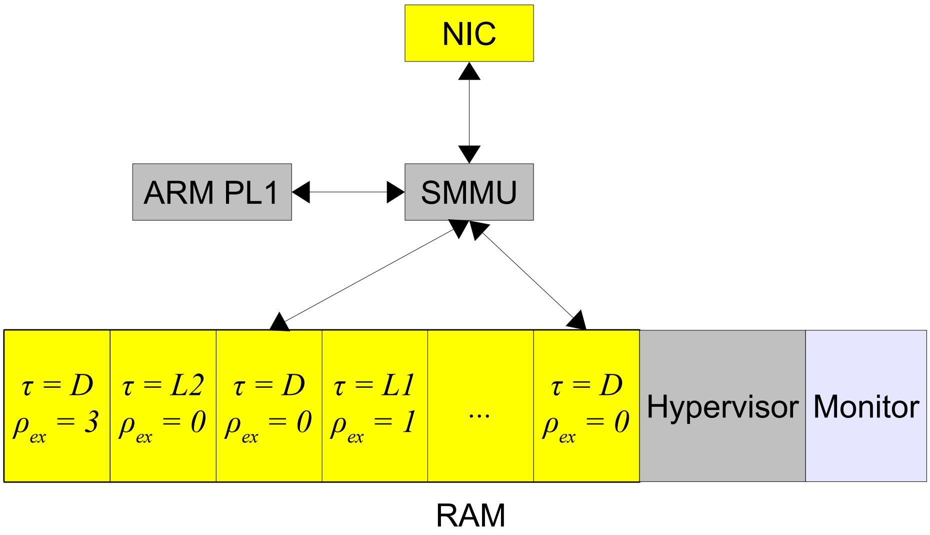

NIC
ARM PL1
SMMU
$\tau = D$ $\rho_{ex} = 3$
$\tau = L2$ $\rho_{ex} = 0$
$\tau = D$ $\rho_{ex} = 0$
$\tau = L1$ $\rho_{ex} = 1$
...
$\tau = D$ $\rho_{ex} = 0$
Hypervisor
Monitor
RAM

Figure 24: How the hypervisor can use an SMMU. The hypervisor configures the system such that the SMMU can only be configured in privileged mode, and therefore the hypervisor is the only software component that can configure the SMMU. The hypervisor then configures the SMMU such that the SMMU prevents the NIC from accessing blocks that are not allocated to Linux, or that are allocated to Linux but contain page tables or are mapped as executable. That is, the NIC can only access non-executable D blocks: blocks bl for which $\tau(bl) = D$ and $\rho_{ex}(bl) = 0$ (indicated by the two bidirectional arrows between the SMMU and the two non-executable D blocks in RAM). When a memory access is performed by the NIC, the access is checked by the SMMU. If the access is to a block of type D and for which ρ_{ex} is equal to zero, the SMMU allows the access and otherwise not. Such a configuration gives Linux direct access to all NIC registers while still ensuring that the NIC cannot transfer the control of the system to Linux or enabling the CPU to execute unsigned Linux code. The SMMU can therefore be used to eliminate the need for the NIC register write request handlers and their associated execution overhead.

additional overhead if the memory accesses performed during this execution cause cache misses when the execution of Linux is resumed.

- An exception return involving 17 reads from memory (or cache) for restoring the contents of the 16 user mode registers and CPSR.

Since the NIC device driver in Linux attempts to perform seven of the nine NIC register writes in consecutive sequence, it is easy to modify the NIC device driver in Linux and the NIC handling code in the hypervisor such that the number of exceptions per transmitted or received frame is reduced from nine to three. Such a reduction can be achieved by implementing a hypercall invocation in the NIC device driver in Linux and a corresponding hypercall in the hypervisor. The hypercall invocation in the NIC device driver in Linux can store all seven values to write and the virtual address of the first NIC register location to write in suitable user mode registers. (One address is sufficient since the seven writes are performed to succeeding addresses.) For each of the seven NIC register writes, the hypercall in the hypervisor invokes the corresponding NIC register write request handler. This implementation reduces only the overhead described in the first and last item bullets. The performance improvement is therefore limited by the overhead described in the second bullet item and which cause the majority of the overhead of NIC register writes.

Since the VE functionality cannot be used to restrict which memory accesses the NIC performs, writes to the NIC registers affecting memory accesses must still be checked. Hence, the NIC register write request handlers and their associated execution overhead cannot be removed despite the usage of VE. In addition to VE, ARM also specifies a hardware device called system memory management unit (SMMU) [37]. The SMMU has a similar relationship to I/O devices as the MMU has to CPUs. The MMU maps virtual addresses accessed by the CPU to physical addresses and prevents accesses that are not permitted, as specified by the page tables located in memory. The SMMU performs equivalent operations according to its own set of page tables, but for memory accesses issued by I/O devices.

The hypervisor can configure a system with an SMMU such that the NIC can only access non-executable *D* blocks as follows:

- The hypervisor configures the system such that the configuration registers of the SMMU and the page tables used by the SMMU are accessible only to the CPU and when the CPU executes the hypervisor.
- The memory mapping request handlers configure the page tables used by the SMMU such that the NIC can only access non-executable *D* blocks.

Such a configuration prevents the NIC from transferring the control of the system to Linux and the CPU from executing unsigned Linux code. Hence, the need for the NIC register write request handlers and their associated execution overhead are eliminated. Figure 24 illustrates this usage of an SMMU.

5 Models

The purpose of the proof plan is to describe how it can be formally proved at the ISA level that only signed Linux code is executed in a physical system consisting of an ARMv7 CPU, a memory, the NIC on BeagleBone Black, the hypervisor, the monitor and Linux. To accomplish such a proof, a formal model must be constructed that describes how the hardware in this physical system executes. The operations performed by the physical CPU must be described by the model at the ISA level, meaning that the model describes each atomic execution step of the physical CPU as the execution of one CPU instruction. The operations performed by the physical NIC are preferably described by the model at a granularity that does not include more operations than one atomic execution step of the physical NIC. Since the property of only signed Linux code being executed depends on memory content, it is critical that the model describes all memory accesses that the physical NIC can perform. Only then can the formal proof imply that only signed Linux code is executed by the physical system. That is, if it is proved on the model that only signed Linux code is executed, then indeed only signed Linux code is executed by the hardware.

Section 5.1 describes a model of the physical NIC that has been designed to take these considerations into account and discusses its correctness.

This NIC model is then used to instantiate the device model framework (described in Subsection 2.4.1) to form a model that describes how hardware consisting of an ARMv7 CPU, a memory, and the NIC on BeagleBone Black executes at the ISA level. The resulting model is called the real model. Section 5.2 describes the real model and discusses its correctness. The proof plan in Chapter 6 describes by means of this model how a formal proof can be constructed of that only signed Linux code is executed.

That proof plan consists of three steps. In the first step it is proved that the software design of the hypervisor and the monitor ensures that only signed Linux code is executed. That first proof step is based on an abstract model describing the software design of the hypervisor and the monitor, called the ideal model. In the second step the simulation proof method is applied by proving on the real model that the execution of the binary code of the hypervisor and the monitor operates according to their software design described by the ideal model. In the third step the property of that only signed Linux code is executed is transferred from the ideal model to the real model. The conclusion is therefore that the hypervisor and the monitor ensure that only signed Linux code is executed by hardware that contains one ARMv7 CPU, a memory, and the NIC on BeagleBone Black. The ideal model is described in Section 5.3.

5.1 Model of the Network Interface Controller

The intent of the NIC model is to guide an implementation in HOL4 of a model of the physical NIC that describes all memory accesses the physical NIC can perform with respect to how the NIC device driver in Linux 3.10 configures the physical NIC. All operations of the physical NIC described in Section 2.3.2 are included in the NIC model. The NIC model is specified in the pseudocode syntax described in Appendix A, and a significant part of that specification is included in Appendix C.

The NIC model is designed to fit the I/O device interface of the device model framework. By basing the proof plan on these two models, the verification results described in this thesis are made as usable as possible for PROSPER, since PROSPER has developed the device model framework and related verification tools.

5.1.1 Semantics of the Model of the Network Interface Controller

To fit the I/O device interface of the device model framework, the NIC model instantiates the four functions of this interface, *d_read*, *d_write*, *progress*, and *receive*, with the NIC model functions *read_nic_register*, *write_nic_register*, *nic_execute* and *memory_byte*, respectively. These four NIC model functions describe the operation of the physical NIC according to the semantics of *d_read*, *d_write*, *progress* and *receive*, described in Subsection 2.4.1, and the NIC specification [32].

The four NIC model functions operate on a state of the NIC model. (The data type of a NIC model state is denoted by nic_state and is partly defined in Subsection C.3). The state of the NIC model contains the values of all variables used by these functions to fulfill their purpose, including the contents of the NIC registers included in the NIC model. The registers of the physical NIC that are included in the NIC model are the ten NIC registers affecting which memory accesses the physical NIC performs. Eight of these registers are shown in Figure 9 in Subsection 2.3.2. The other two NIC registers are DMACONTROL and RX_BUFFER_OFFSET. All ten of these NIC registers are described in Section C.1.

Since it is only necessary for the NIC model to describe the memory accesses the physical NIC performs considering how the NIC device driver in Linux 3.10 configures the physical NIC, nearly all other operations performed by the physical NIC are not described by the NIC model. In addition, for natural reasons, physical NIC operations unspecified (not described) by the NIC specification are also not described by the NIC model. To make the NIC model sound, when the NIC model shall describe a physical NIC operation that is either not included in the NIC model or that is unspecified by the NIC specification, the four NIC model functions return a NIC model state that is marked as dead. A dead state is an undefined state that cannot be left.

The NIC model is a transition system with four types of transitions: register read, register write, autonomous, and memory read request reply transitions, described by *read_nic_register*, *write_nic_register*, *nic_execute* and *memory_byte*, respectively. The following list summarizes how these four functions describe the operation of the physical NIC:

- (nic_state, word32) *read_nic_register*(nic_state *nic*, word32 *pa*)

  Describes a register read transition of the NIC model. Given a state of the NIC model, *nic*, and a physical address of the NIC register to read, *pa*, this function returns an updated state of the NIC model and the 32-bit value of the read NIC register in the state *nic*.

If *pa* is not 32-bit word aligned, then the returned state is marked as dead. The reason for marking the returned NIC state as dead is because the ARMv7 specification does not allow unaligned accesses to I/O device registers [33, p. 109].

If *pa* is 32-bit word aligned, then the returned state is the same as the input state *nic*, since reads of the ten NIC registers included in the NIC model do not cause the physical NIC to perform any operation. If one of the NIC registers included in the NIC model is accessed, the value of that register as recorded by *nic* is returned. If a NIC register not included in the NIC model is accessed, a non-deterministically chosen value is returned, since the value of the accessed register is unknown.

- nic_state *write_nic_register*(nic_state *nic*, word32 *pa*, word32 *value*)

  Describes a register write transition of the NIC model. Returns the NIC model state that the NIC model enters from the state *nic* when the value *value* is written to the NIC register located at physical address *pa*. If *pa* is not 32-bit word aligned or such a write causes the physical NIC to perform an unspecified operation, the returned state is marked as dead. Otherwise, the state of the NIC model is updated to reflect the reaction of the physical NIC when such a write is performed.

- (nic_state, mem_req ∪ {⊥}, bool) *nic_execute*(nic_state *nic*)

  Describes an autonomous transition of the NIC model. The purpose of *nic_execute* is to describe one execution step of the physical NIC. To allow detailed reasoning of the behavior of the physical NIC, *nic_execute* considers an execution step of the physical NIC to be one fine-grained hardware operation. Each such operation is either one access to one byte in memory or in CPPI_RAM, or one access to one field of a NIC register or a buffer descriptor in CPPI_RAM.

  The first component of the returned tuple is some state (non-deterministic function as explained later in Subsection 5.1.1.1) that the NIC model enters when it performs one autonomous transition from the state *nic*. If the NIC model does not describe the operation to be performed, the returned state is marked as dead.

  The second component contains information related to memory requests issued by the NIC model. If the returned value is equal to ⊥, the NIC model made no memory request. Otherwise, the data type mem_req contains information of whether it is a read or write request, the physical address of which memory byte to access, and if it is a write, which value to write. *nic_execute* never issues a memory read request if an earlier one has not been replied.

  The third component is true if and only if the NIC model in the returned state is asserting an interrupt, according to the value of an interrupt flag in that returned state. The interrupt flag in a state is non-deterministically set to true by *nic_execute* when *nic_execute* writes a CP register. The reason for this non-deterministic behavior is because the interrupt related NIC registers are not included in the NIC model and it is therefore unknown whether interrupts are enabled or not. By setting the interrupt flag to true non-deterministically, the NIC model describes the operation of the physical NIC with respect to both enabled interrupts and disabled interrupts.

- nic_state *memory_byte*(nic_state *nic*, mem_req *reply*)

  Describes memory read request reply transitions. If the NIC model in the state *nic* does not expect the memory read request reply *reply*, the returned state is marked as dead. *memory_byte* considers a reply as expected if the following two conditions are satisfied. The first condition is that the value of the flag *memory_request* in the state *nic* is true. It is set to true if and only if a memory read request has been issued and not replied. The second condition is that the physical address of the last issued memory read request by the NIC model (that address is stored in *nic*), is equal to the physical address specified by *reply*. If the reply is expected as determined by these two conditions, then the returned state is equal to the state *nic* but with the flag *memory_request* set to false, reflecting that the last issued memory read request has been replied. Since the operation of the physical NIC does not depend on memory content, the read byte value specified by *reply* is ignored.

5.1.2 Autonomous Transitions of the NIC Model

This subsection describes the implementation of *nic_execute* in deeper detail, which constitutes the largest part of the NIC model. The other parts of the NIC model, *read_nic_register*, *write_nic_register* and *memory_byte*, are specified in pseudocode with comments in Sections C.4, C.5 and C.8, respectively.

By considering which operations the physical NIC performs as a result of how software configures it, as described in Subsections 2.3.2.1 through 2.3.2.5, the physical NIC can be viewed as performing five tasks: initialization of itself, transmission of frames, reception of frames, tear down of transmission of frames, and tear down of reception of frames. The execution steps of the physical NIC are therefore naturally described by *nic_execute* by means of five automata. Each automaton describes how the physical NIC performs one of these five tasks. Each autonomous transition of the NIC model described by *nic_execute* is one transition of one of these five automata. Which automaton that shall perform the next autonomous transition of the NIC model is non-deterministically decided by *nic_execute*. Figure 25 illustrates this idea.

Each of the five tasks the physical NIC performs is either idle, active or pending. A task is idle when the NIC does not perform it, active when the NIC performs it, and pending if the NIC will perform it but must complete another task before performing it. Each automaton is therefore either idle, active or pending. When the NIC is powered on, all task are idle. All automata are therefore initially idle. *nic_execute* cannot select an automaton that is idle or pending to perform the next transition of the NIC model, since such an automaton cannot perform a transition.

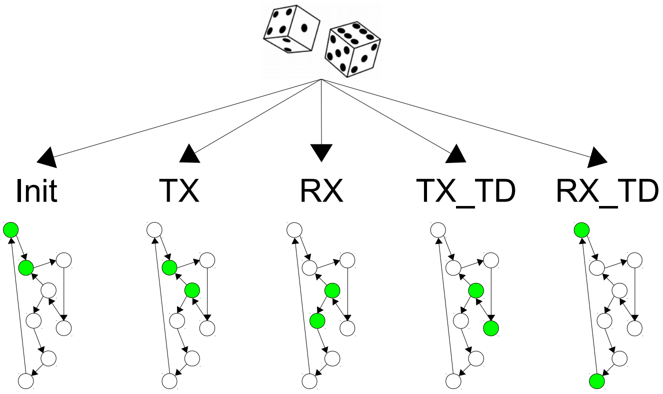


Figure 25: The autonomous transitions of the NIC model are performed by five automata. The five automata describe the operations that are related to initialization (Init), transmission (TX), reception (RX), transmission teardown (TX_TD) and reception teardown (RX_TD). Which automaton that shall perform the next autonomous transition of the NIC model is decided non-deterministically by nic_execute. That is, the automaton that shall perform the next autonomous transition of the NIC model is selected arbitrarily. A simplified transition system of the NIC model is shown below each label of each automaton. A transition between a pair of highlighted states denotes the next autonomous transition the NIC model performs if the corresponding automaton is selected by nic_execute. Note that the current state of the NIC model is not the same in all five shown transition systems. The current state of the NIC model is the same only in the two transition systems shown below TX and RX. The next transitions of the transmission and reception automata are therefore starting from the same state.

(If no automata is active when *nic_execute* is applied by the device model framework function *advance_single*, *nic_execute* returns the argument state.) When the CPU writes a NIC register, that write might cause the NIC to initiate a task. The initiated task becomes active if the NIC can perform it, and pending if the NIC must complete another task before performing it. The automaton describing the initiated task therefore becomes active or pending, respectively. When the NIC completes a task, that task becomes idle. The automaton describing the completed task therefore becomes idle. If the NIC can perform a pending task as a result of completing another task, the pending task becomes active. The pending automaton describing the pending task therefore becomes active. Hence, some automata depend on other automata.

The following list summarizes which operations of the physical NIC each automaton describes, when each automaton becomes active and idle, and which relationship each automaton has with the other automata with respect to being idle, active or pending:

- The initialization automaton: Describes the initialization operations of the physical NIC. This automaton becomes active or pending when the CPU model of the device model framework sets bit zero of the CPDMA_SOFT_RESET register. This automaton becomes idle after it has cleared that bit and thereafter the CPU model has initialized the HDP and CP registers to zero.

This automaton is only active (and can perform transitions) when all other automata are idle. There are three reasons:

  - Initiation of the transmission, reception or teardown tasks while the physical NIC performs the initialization task is described by the NIC model as entering a dead state, since it is unspecified how the physical NIC operates in these cases.
  - Initiation of the initialization task while the physical NIC performs a teardown task is described by the NIC model as entering a dead state, since it is unspecified how the physical NIC operates in this case.
  - The initialization task is pending until the physical NIC has completed the transmission of the frame currently being transmitted, and similarly for reception. Hence, the initialization automaton cannot become active until the transmission and reception automata have transmitted and received their currently processed frames and become idle.

Hence, if the initialization automaton is active, no other automaton can become active. If any other automaton is active, the initialization automaton can only become pending if the NIC model does not enter a dead state.

- The transmission automaton: Describes the processing of the queue of buffer descriptors in transmission DMA channel zero. (Since Linux only uses transmission DMA channel zero for transmission, the NIC model only describes transmission DMA channel zero, and not all eight transmission DMA channels.) This automaton becomes active when the CPU model writes the TX0_HDP register with the physical address of the first buffer descriptor in the queue to process. This automaton becomes idle either after all buffer descriptors in the queue have been processed, or after the transmission of the current frame is complete and the initialization or transmission teardown automaton is pending.

  The transmission automaton is only active (and can perform transitions) when the initialization and transmission teardown automata are idle or pending. There are two reasons:

  - The initialization and teardown tasks are pending until the physical NIC has completed the transmission of the frame currently being transmitted. Hence, the initialization and transmission teardown automata cannot become active until the transmission automaton has transmitted the current frame and become idle.
  - Initiation of the transmission task while the physical NIC performs the initialization or transmission teardown task is described by the NIC model as entering a dead state, since the behavior of the physical NIC in that case is unspecified.

Hence, if the transmission automaton is active, the initialization and transmission teardown automata cannot become active. If the initialization or the transmission teardown automaton is active and the transmission automaton is to become active, the NIC model enters a dead state.

- The reception automaton: Describes the processing of the queue of buffer descriptors in reception DMA channel zero. These buffer descriptors are used to store the most recently received frame. (As for transmission, since Linux only uses reception DMA channel zero for reception, the NIC model only describes reception DMA channel zero and not all eight reception DMA channels.)

  The reception automaton becomes active when *nic_execute* has non-deterministically decided that a new frame has been received. That occurs when the following five conditions hold:

  - The reception automaton is idle, meaning the physical NIC is free to process a new received frame.
  - There are unused buffer descriptors in reception DMA channel zero, meaning there is memory allocated for the physical NIC to store received frames.
  - The initialization automaton is idle and the current NIC model state reflects a physical NIC that has been initialized, meaning the physical NIC is not currently being initialized but has been initialized.
  - The reception teardown automaton is idle, meaning reception DMA channel zero of the physical NIC is not being teared down.
  - *nic_execute* selects the reception automaton to perform a transition, meaning *nic_execute* decided that a frame has been received.

  The first four conditions describe a state of the physical NIC in which the physical NIC can process a received frame, and the last condition describes the case where the physical NIC has received a frame which the physical NIC will process. This means that the reception task of the physical NIC becomes active, and therefore the reception automaton also becomes active.

  The reception automaton becomes idle when the received frame has been stored in memory and all of its associated buffer descriptors have been processed. For similar reasons as for the transmission automaton, this automaton is only active (and can perform transitions) when the initialization and reception teardown automata are idle or pending.

- The transmission teardown automaton: Describes the operations that the physical NIC performs to tear down transmission DMA channel zero. This automaton becomes active when the CPU model writes zero to the TX_TEARDOWN register. (Since the NIC model only describes transmission DMA channel zero, only zero can be written to this register and writes of any other value causes the NIC model to enter a dead state.) This automaton becomes idle after it has written some buffer descriptor fields in the first unused buffer descriptor in reception DMA channel zero, cleared the TX0_HDP register, and written the TX0_CP register with 0xFFFFFFFC to generate a transmission teardown interrupt.

This automaton is only active (and can perform transitions) when the initialization and transmission automata are idle. There are three reasons:

  - Initiation of the initialization task while the physical NIC performs the transmission teardown task, and vice versa, is described by the NIC model as entering a dead state, since the behavior of the physical NIC in those cases is unspecified.
  - Initiation of the transmission task while the physical NIC performs the transmission teardown task is described by the NIC model as entering a dead state, since the behavior of the physical NIC in this case is unspecified.
  - The transmission teardown task is pending until the physical NIC has completed the transmission of the frame currently being transmitted. Hence, the transmission teardown automaton cannot become active until the transmission automaton has transmitted its currently processed frame and become idle.

  Hence, if the NIC model does not enter a dead state when describing each of these scenarios, it is because the transmission teardown automaton becomes pending or active when the transmission automaton is active or idle, respectively.

- The reception teardown automaton: Similar to the transmission teardown automaton but with respect to reception.

5.1.2.1 Non-Deterministic Selection of NIC Automaton Transitions

In the initial state of the NIC model, all automata are idle, reflecting a physical NIC that has just been powered on. The automata become active or pending when the CPU model writes certain NIC registers under certain conditions. When the scheduler of the device model framework determines that the NIC model shall perform an autonomous transition, *advance_single* of the device model framework applies *nic_execute*. If several automata are active, *nic_execute* must decide which active automaton shall perform that next autonomous transition of the NIC model. *nic_execute* makes that decision non-deterministically. Hence, the NIC model has its own non-deterministic scheduler, which schedules automata. The execution of the physical NIC is therefore described by the NIC model as the set of all possible interleavings of the transitions performed by the five automata. Figure 26 shows an execution trace of the NIC consisting of a sequence of autonomous NIC transitions.

nic_execute selects the automaton that shall perform the next autonomous transition of the NIC model as follows. First, *nic_execute* forms a set *A* containing identifiers of all active automata, except for the initialization automaton if it has performed the reset operation (as explained in Subsection 5.1.2.2). If the reception automaton is idle, and the current NIC model state satisfies the middle three

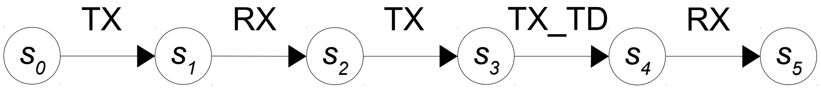
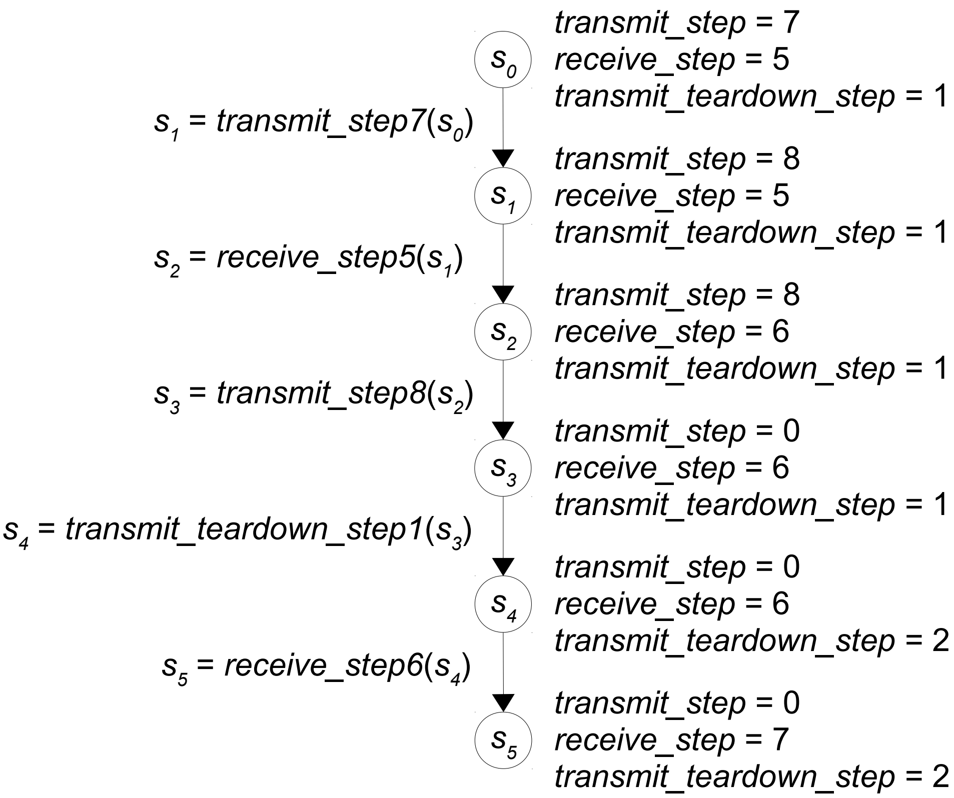


Figure 26: An execution trace consisting of autonomous NIC transitions. TX is an abbreviation for the transmission automaton, RX for the reception automaton, and TX_TD for the transmission teardown automaton. It is assumed that the transmission and reception automata are active and the transmission teardown automaton is pending in state s_0. In this example, nic_execute has decided that first shall the transmission automaton make a transition and then the reception automaton. Thereafter, the transmission automaton performs its last transition and becomes idle in state s_3. The transmission teardown automaton therefore becomes active, enabling it to be selected by nic_execute to perform its first transition the next time nic_execute is applied by the device model framework function advance_single. For the last two transitions, nic_execute first selects the transmission teardown automaton and then the reception automaton, leaving the NIC model in state s_5.

conditions listed in Subsection 5.1.2 in the bullet item for the reception automaton, the identifier of the reception automaton is also added to *A*. Then, *nic_execute* non-deterministically selects an identifier in *A*. The selected identifier identifies which automaton that shall perform the next autonomous transition of the NIC model. If *A* is empty, *nic_execute* returns the argument state, meaning the autonomous NIC transition did nothing.

When *nic_execute* has selected an automaton to perform the next autonomous transition of the NIC model, *nic_execute* applies the transition function of that automaton. That automaton has a set of step functions, and the transition function applies one of those step functions. To determine which step function to apply, each automaton has a nonnegative step variable, whose value is stored in the state of the NIC model. The value of the step variable identifies which step function the transition function shall apply. The application of the identified step function makes the selected automaton perform its next transition. That transition is also the next autonomous transition of the NIC model.

Furthermore, if a step variable of an automaton is equal to zero, the automaton is idle, if the step variable is equal to one, the automaton is active or pending, and if the step variable is greater than one, the automaton is active. An automaton is pending if and only if it must wait for another automaton to become idle. For instance, assume the step variable of the reception teardown automaton is one. If the step variable of the reception automaton is zero, the reception teardown automaton is active, since this means that the reception task of the physical NIC is idle and therefore the physical NIC can perform the reception teardown task. If the step variable of the reception automaton is nonzero, the reception teardown automaton is pending, since this means that the physical NIC performs the reception task and therefore the physical NIC must wait with performing the reception teardown task. Figure 27 and the following three subsections (5.1.2.2 through 5.1.2.4) illustrate the use of step variables and step functions, and how automata transition between being idle, active and pending.



transmit_step = 7
receive_step = 5
transmit_teardown_step = 1
s_0
s_1 = transmit_step7(s_0)
transmit_step = 8
receive_step = 5
transmit_teardown_step = 1
s_1
s_2 = receive_step5(s_1)
transmit_step = 8
receive_step = 6
transmit_teardown_step = 1
s_2
s_3 = transmit_step8(s_2)
transmit_step = 0
receive_step = 6
transmit_teardown_step = 1
s_3
s_4 = transmit_teardown_step1(s_3)
transmit_step = 0
receive_step = 6
transmit_teardown_step = 2
s_4
s_5 = receive_step6(s_4)
transmit_step = 0
receive_step = 7
transmit_teardown_step = 2
s_5

Figure 27: An example of how the step functions and step variables of each automaton are used to perform automaton and autonomous NIC transitions. The step function applications on the left side perform the transitions next to them and modify the step variables of the states as indicated by the variable values on the right side of the states. In state s_0, nic_execute selects the transmission automaton to perform the next autonomous transition of the NIC model. Since the step variable of the transmission automaton, transmit_step, is equal to seven, the transition function of the transmission automaton applies step function seven of the transmission automaton, transmit_step7, which sets transmit_step to eight. Next time nic_execute selects the transmission automaton, the transition function of the transmission automaton will apply step function eight of the transmission automaton, transmit_step8. In this example, nic_execute selects the reception automaton before nic_execute selects the transmission automaton again. The transmission automaton is active in states s_0, s_1 and s_2, since its step variable is nonzero in those states and the transmission automaton never waits for other automata to become idle. The transmission teardown automaton is pending in states s_0, s_1 and s_2, since its step variable is equal to one and it must wait for the transmission automaton to become idle. After transmit_step8 has been applied and set transmit_step to zero, the transmission automaton is idle, since its step variable is set to zero. The transmission teardown automaton therefore becomes active. This enables nic_execute to select the transmission teardown automaton to perform the next autonomous NIC transitions, which nic_execute does when given state s_3. The transition function of the transmission teardown automaton then applies transmit_teardown_step1, since the step variable transmit_teardown_step is equal to one. The usage of step functions and step variables is identical for all automata.

Apart from having a step variable, each automaton might also have additional variables that are accessed by its step functions. The values of those variables are recorded in the state of the NIC model.

The following three subsections describe the initialization, transmission and transmission teardown automata in deeper detail. Since the operations related to reception are similar to those of transmission, and the latter operations are fewer, the descriptions of the reception and reception teardown automata are omitted. The differences are that the transmission automaton reads bytes in memory while the reception automaton writes them, and the usage of certain buffer descriptor fields.

5.1.2.2 Initialization Automaton

Figure 28 shows the initialization automaton, which operates as follows, assuming it is initially idle. If the CPU model sets bit zero of the CPDMA_SOFT_RESET register, and a step variable of some teardown automaton is nonzero, meaning some teardown automaton is active, *write_nic_register* sets the NIC model in a dead state (since it is unspecified which operations the physical NIC performs when a reset operation of the DMA hardware is initiated while a DMA channel is being teared down). If both step variables of the teardown automata are zero, meaning they are idle, *write_nic_register* sets the step variable of the initialization automaton to one. If both step variables of the transmission and reception automata are zero, meaning they are idle, the initialization automaton becomes active. Otherwise, the initialization automaton becomes pending. When the step functions of the transmission and reception automata have set the step variables of those two automata to zero, meaning they have become idle, the initialization automaton has become active. When the initialization automaton is active, it can be selected by *nic_execute* to perform its next transition.

Next time *nic_execute* determines that the initialization automaton shall perform a transition, *nic_execute* applies the transition function of the initialization automaton. That transition function clears bit zero of the CPDMA_SOFT_RESET register and sets the step variable of the initialization automaton to two. (Since the initialization automaton only performs these two operations, and in one single transition, its only step function is instead implemented by its transition function.) The execution of these two operations means that the physical NIC has performed the reset operation of the DMA hardware, and that the physical NIC now waits for the physical CPU to initialize the HDP and CP registers. Even though the initialization automaton is active, since its step variable is nonzero, *nic_execute* will not select it to perform additional transitions. The reason is that the automaton has performed its operations, and it is the task of the software to initialize the HDP and CP registers.

When the CPU model has initialized the TX0_HDP, RX0_HDP, TX0_CP and RX0_HDP registers to zero, the NIC model reflects a physical NIC that has been initialized. (The NIC model only requires the HDP and CP registers of DMA channels zero to be initialized since Linux only uses them.) *write_nic_register* therefore sets the step variable of the initialization automaton to zero at that point, meaning the initialization automaton becomes idle.

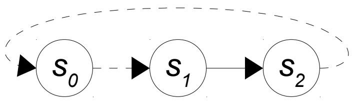


Figure 28: The initialization automaton. Each circle represents an automaton state of the initialization automaton. An automaton state represents all states in the NIC model that have a specific value of the step variable of the corresponding automaton. The initialization automaton has three automaton states, s_0, s_1 and s_2, since its step variable has three possible values, zero, one and two. Automaton states s_0, s_1 and s_2 represents the states in the NIC model in which the value of the step variable of the initialization automaton is equal to zero, one and two, respectively. In the NIC model states represented by s_0, the initialization automaton is idle. In the NIC model states represented by s_1, the initialization automaton is active or pending. If the step variables of the transmission and reception automata are zero in a NIC model state represented by s_1, meaning they are idle in that NIC model state, then the initialization automaton is active in that NIC model state. Otherwise, the initialization automaton is pending in that NIC model state. In the NIC model states represented by s_2, the initialization automaton is active. The transitions show how the step variable of the initialization automaton is modified. The transition represented by the continuous arrow is described by the transition function of the initialization automaton (which implements its only step function), meaning that the initialization automaton itself modifies its step variable in that transition. The transitions represented by dashed arrows are described by write_nic_register, meaning that the initialization automaton itself does not modify its step variable in those transitions. That is, these latter modifications of the step variable are not caused by the initialization automaton performing an autonomous NIC transition, but instead caused by the CPU model performing a transition. The transition from s_0 to s_1 occurs when the CPU model sets bit zero of the CPDMA_SOFT_RESET register, and is described by write_nic_register. The transition from s_1 to s_2 occurs when the initialization automaton performs the reset operation of the DMA hardware. That reset operation is described by the transition function of the initialization automaton, which clears bit zero of the CPDMA_SOFT_RESET register. The transition from s_2 to s_0 is not a single transition of the NIC model. It represents four transitions of the NIC model, each of which occurs when the CPU model writes zero to the TX0_HDP, RX0_HDP, TX0_CP or RX0_CP register, respectively. The transition from s_2 to s_0 occurs when the CPU model has written zero to all these four registers, and is described by write_nic_register.

5.1.2.3 Transmission Automaton

Figure 29 shows the transmission automaton, which operates as follows, assuming it is initially idle. When the NIC model is in a state where the initialization automaton either has not performed all of its initialization operations or is not idle, the NIC model reflects a physical NIC which has not been initialized or which is currently being initialized. When the physical NIC has not been initialized or is being initialized, it is unspecified how the physical NIC operates when the physical CPU writes the TX0_HDP register to start transmission. *write_nic_register*

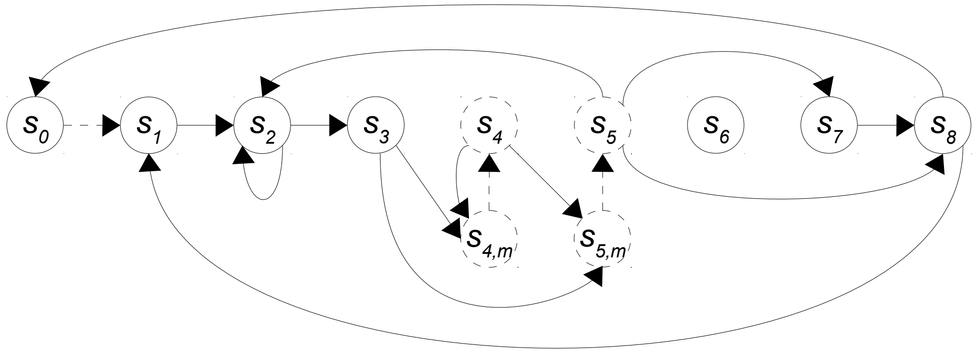


Figure 29: The transmission automaton. The continuous circles labeled s_0 through s_3 and s_6 through s_8 represent automaton states of the transmission automaton (see the caption of Figure 28 for an explanation of automaton states). The dashed circles labeled s_4 and $s_{4,m}$ together represent the automaton state where the step variable of the transmission automaton is equal to four, and similarly for s_5 and $s_{5,m}$ but with the step variable equal to five. The transitions represented by continuous arrows are described by the step functions of the transmission automaton, while the transitions represented by dashed arrows are described by write_nic_register or memory_byte. A cyclic sequence of transitions starting from and ending at s_0 processes all buffer descriptors in transmission DMA channel zero. The transition from s_0 to s_1 activates the transmission automaton and occurs when the CPU model writes the TX0_HDP register. The transition from s_1 to s_2 checks that the next buffer descriptor to process is located in CPPI_RAM at a 32-bit word aligned physical address. Each transition from s_2 reads one byte of the next buffer descriptor to process from CPPI_RAM. The transitions from s_3 and s_4 to $s_{4,m}$ and $s_{5,m}$ issue memory read requests, and set the NIC model variable memory_request to true (thereby the letter m in the index of $s_{4,m}$ and $s_{5,m}$). In $s_{4,m}$, there are additional memory read requests to issue for the bytes of the data buffer addressed by the current buffer descriptor, while in $s_{5,m}$, all memory read requests have been issued. When the device model framework applies the NIC model function memory_byte to reply to a memory read request (see Subsection 2.4.1), memory_byte sets memory_request to false and the transmission automaton transitions from $s_{4,m}$ to s_4 or from $s_{5,m}$ to s_5. If the current frame under transmission is located in several data buffers addressed by several buffer descriptors (as shown in Figure 10 in Subsection 2.3.2.2), and all of those buffer descriptors have not been processed, the next buffer descriptor in the queue shall be processed. In such a case, the transmission automaton transitions from s_5 to s_2 (that transition also includes the operations of the transition from s_1 to s_2). This means that a sequence of transitions starting from s_1 and ending at s_5 processes a single buffer descriptor. When all buffer descriptors of the current frame under transmission has been processed, the transmission automaton writes some NIC registers and fields of the buffer descriptors addressing the data buffers of that frame. These writes are performed by the transitions from s_5, s_7 and s_8. If there are no additional frames to transmit, the transmission automaton transitions from s_8 to s_0 and becomes idle. Otherwise, it transitions from s_8 to s_1 to process the first buffer descriptor of the next frame to transmit. There is no transition to s_6 (and thereby no transition from s_6) since step function five either applies step function one or six, which cause the step variable to be set to two, or to seven or eight, respectively.

therefore sets the NIC model in a dead state if the CPU model writes TX0_HDP when the NIC model is in a state where the initialization automaton either has not initialized the NIC or is not idle.

If the CPU model writes TX0_HDP when the initialization automaton has performed all of its initialization operations and is idle, *write_nic_register* sets the step variable of the transmission automaton to one. The transmission automaton therefore becomes active. Since the transmission automaton never waits for any other automaton to become idle before the transmission automaton can perform its transitions, the transmission automaton is never pending. The value written to TX0_HDP is the physical address of the first buffer descriptor in the queue that is given to transmission DMA channel zero. The processing of that channel is described by the transmission automaton by means of its eight step functions. Those eight step functions describe the operation of the transmission automaton as follows:

1. Checks that the address of the currently processed buffer descriptor is 32-bit word aligned and that all of its sixteen bytes are located in CPPI_RAM. (In the following seven step function descriptions, this current buffer descriptor will only be referred to as the buffer descriptor.) This buffer descriptor is the head of the buffer descriptor queue in transmission DMA channel zero when the transmission automaton becomes active. If the check fails, the current NIC model state is marked as dead. Otherwise, step function two is applied, since this first step function performs no hardware operation (that is, not accessing the memory or a NIC register; for an explanation of hardware operations, see the description of the function *nic_execute* in Subsection 5.1.1).
2. Reads the next byte of the buffer descriptor from CPPI_RAM. When all 16 bytes of it have been read, the step variable is set to three. The next time the transmission automaton is selected by *nic_execute*, the transition function of the transmission automaton applies step function three of the transmission automaton.
3. Checks that certain fields of the buffer descriptor are correctly initialized. If the check fails, the state is marked as dead. Otherwise, step function four is applied.
4. Issues the next memory read request for the next byte to read of the data buffer addressed by the buffer descriptor. To keep the NIC model simple, the transmission automaton reads the bytes of the data buffer sequentially and the transmission automaton does not issue an additional memory read request until the pending one has been replied. To prevent *nic_execute* from selecting the transmission automaton to perform its next transition, and thereby potentially issue an additional memory read request, while a memory read request is pending, the transmission automaton has a NIC model variable *memory_request* that is either true or false.

   When this step function issues a memory read request, it sets *memory_request* to true. When *memory_request* is true, *nic_execute* does not select the transmission automaton to perform its next transition. When the device model framework gives the content of the byte at the physical address specified by the issued memory read request, the device model framework function *advance_single* applies the device model framework function *mem_acc_by_dev*, which in turn applies the NIC model function *memory_byte* (see Subsection 2.4.1 where *memory_byte* instantiates *receive*). *memory_byte* sets *memory_request* to false, since the issued memory read request has been replied. When *memory_request* is false, *nic_execute* can select the transmission automaton again to perform its next transition. If there are additional memory read requests to issue, the step variable of the transmission automaton is equal to four. The transition function of the transmission automaton therefore applies this fourth step function, which then issues the next memory read request.

When this step function issues the final memory read request to read the last byte of the data buffer addressed by the buffer descriptor, it sets *memory_request* to true and the step variable to five. This causes the device model framework to reply to this final memory read request, by applying *memory_byte*, which sets *memory_request* to false. Since *memory_request* is false, the next time the device model framework applies *nic_execute*, *nic_execute* can select the transmission automaton. If the transmission automaton is selected, its transition function applies step function five.

5. Now when all bytes of the data buffer addressed by the buffer descriptor has been read from memory, it is checked whether the complete frame has been read. If that is the case, step function six is applied. Otherwise, step function one is applied to process the next buffer descriptor in the queue. That next buffer descriptor addresses the data buffer containing the next part of the frame. Since this fifth step function does not describe a hardware operation, it ends by applying either step function one or six.

6. If the buffer descriptor is the last one in the queue, its end of queue bit is set (see Subsection 2.3.2.2 for a description of buffer descriptor fields), and the step variable is set to seven. Otherwise, no hardware operation is described by this sixth step function, and therefore it applies step function seven.

7. The ownership bit is cleared in the buffer descriptor addressing the first part of the read frame. If the current buffer descriptor, which addresses the last part of the read frame, is last in transmission DMA channel zero, TX0_HDP is set to zero. Finally, the step variable is set to eight.

8. The physical address of the current buffer descriptor is written to TX0_CP. Such a write generates a transmission completion interrupt, if such interrupts are enabled. Such an interrupt signals that the read frame has been transmitted. The state of the NIC model includes a boolean variable that is true if and only if the NIC model in its current state describes a physical NIC that is currently asserting a transmission completion interrupt. This variable is set to true non-deterministically. That is, sometimes it is set to true and sometimes it is not set at all. The reason for setting it non-deterministically is because (i) the interrupt control registers of the physical NIC are not described by the NIC model, and (ii) setting it non-deterministically allows the NIC model to describe the operation of the

physical NIC for both cases where transmission completion interrupts are enabled and disabled. Section C.10 describes the interrupt control registers of the physical NIC that must be described by the NIC model in order for the NIC model to describe physical NIC interrupts deterministically.

If all buffer descriptors in transmission DMA channel zero have now been processed or the transmission teardown or initialization automata are pending (some of their step variables is equal to one), the step variable of the transmission automaton is set to zero to indicate that the transmission automaton is idle. Otherwise the step variable is set to one to indicate that the next time the transmission automaton is selected by *nic_execute* to perform the next transition of the NIC model, the transmission automaton shall begin to process the next buffer descriptor in transmission DMA channel zero. That buffer descriptor addresses the first part of the next frame to transmit.

5.1.2.4 Transmission Teardown Automaton

Figure 30 shows the transmission teardown automaton, which operates as follows, assuming it is initially idle. If the CPU model writes TX_TEARDOWN to a nonzero value, or if the CPU model writes TX_TEARDOWN when the initialization automaton either has not initialized the NIC or is not idle, the NIC model enters a dead state (since in this scenario the operations performed by the physical NIC are unspecified). Otherwise, when the CPU model writes TX_TEARDOWN to zero and the initialization automaton both has initialized the NIC and is idle, *write_nic_register* sets the step variable of the transmission teardown automaton to one. If the transmission automaton is idle, the transmission teardown automaton becomes active. Otherwise the transmission teardown automaton becomes pending. The transmission teardown automaton remains pending until the transmission automaton becomes idle, which occurs when step function eight of the transmission automaton sets the step variable of the transmission automaton to zero. At that point the transmission teardown automaton becomes active, and therefore also selectable by *nic_execute* to perform its transitions.

The operations of those transitions are described by the four step functions of the transmission teardown automaton as follows:

1. Sets the step variable to two. If transmission DMA channel zero contains buffer descriptors, meaning the transmission automaton has not processed all buffer descriptors in that channel, the end of queue bit of the first buffer descriptor in that channel is set non-deterministically (either it is written to one or not written at all). This bit is set non-deterministically since the specification does not explicitly specify this behavior even though the hardware on BeagleBone Black sets it. In the other case where transmission DMA channel zero does not contain any buffer descriptors, step function two is applied.
2. Sets the step variable to three. If transmission DMA channel zero contains buffer descriptors, the teardown completion bit is set in the first buffer descriptor in that channel. Otherwise step function three is applied.

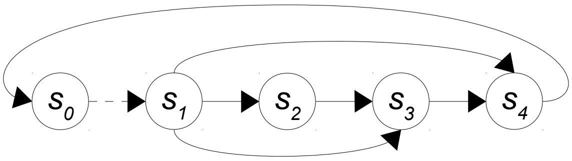


Figure 30: The transmission teardown automaton. Each circle represents an automaton state of the transmission teardown automaton (see the caption of Figure 28 for an explanation of automaton states). In the NIC model states represented by s_0, the transmission teardown automaton is idle. In the NIC model states represented by s_1, the transmission teardown automaton is active or pending. The transmission teardown automaton is active in a NIC model state represented by s_1 if the step variable of the transmission automaton is zero in that NIC model state. Otherwise the transmission teardown automaton is pending in that NIC model state. In the NIC model states represented by s_2, s_3 and s_4, the transmission teardown automaton is active. The transitions show how the step variable of the transmission teardown automaton is modified. The transition from s_0 to s_1, represented by a dashed arrow, is described by write_nic_register. The other transitions, represented by continuous arrows, are described by the step functions of the transmission teardown automaton. The transition from s_0 to s_1 occurs when the CPU model writes zero to the TX_TEARDOWN register. If transmission DMA channel zero contains buffer descriptors when the transmission teardown automaton becomes active, the transmission teardown automaton will write certain buffer descriptor fields of the first buffer descriptor in that channel. (The buffer descriptors that exist in this channel, when the transmission teardown automaton becomes active, have not been processed by the transmission automaton.) The writes to the first buffer descriptor in transmission DMA channel zero are described by the transitions from s_1 to s_2, from s_1 to s_3, from s_2 to s_3 and from s_3 to s_4. If the transmission teardown automaton in a NIC model state represented by s_1 decides to set the end of queue bit in the buffer descriptor (which is decided non-deterministically), the transitions from s_1 to s_2 and from s_2 to s_3 occur. These two transitions set the end of queue and teardown completion bits of the buffer descriptor, respectively. Otherwise the transition from s_1 to s_3 occurs, which just sets the teardown completion bit of the buffer descriptor. The transition from s_3 to s_4 occurs when the transmission teardown automaton clears the TX0_HDP register and the ownership bit in the buffer descriptor. If transmission DMA channel zero contains no buffer descriptors when the transmission teardown automaton becomes active (the transmission automaton has processed all buffer descriptors in that channel), the transmission teardown automaton will instead only perform the transition from s_1 to s_4. The transition from s_1 to s_4 only clears the TX0_HDP register and accesses no buffer descriptor, since there is none. Finally, the transition from s_4 to s_0 occurs when the transmission teardown automaton writes the TX0_CP register with the value 0xFFFFFFFC (the teardown interrupt code).

3. Sets the step variable to four and clears the TX0_HDP register. If transmission DMA channel zero contains buffer descriptors, the ownership

bit is cleared in the first buffer descriptor in that channel, indicating that the buffer descriptor is not in use by the NIC any more.

4. Sets the step variable to zero, the TX0_CP register to 0xFFFFFFFC, and the corresponding interrupt variable to true non-deterministically (either set to true or not set at all).

5.1.3 Accuracy of the NIC Model

This section discusses the accuracy of the NIC model by focusing on the main potential issues:

- Incorrect modeling of the physical NIC due to potential bugs in the NIC model.
- Omitted behavior of the physical NIC related to memory accesses due to potential overspecification of the atomicity and order of certain operations of the physical NIC. Overspecification in this context means that the NIC model describes behavior of the physical NIC that is not stated in the NIC specification. For instance, in the NIC model the transmission teardown automaton first sets the end of queue bit and then the teardown completion bit (see the descriptions of step functions one and two of the transmission teardown automaton in Subsection 5.1.2.4). However, the physical NIC might perform these two operations in the opposite order.

5.1.3.1 Potential Bugs in the NIC Model

The main source for introducing bugs in the NIC model is misunderstandings of the NIC specification. The NIC specification is in general informal and vague. Two issues encountered in the NIC specification are contradictions (e.g. whether a certain bit is set in the buffer descriptor addressing the first part of a frame or in the buffer descriptor addressing the last part of the same frame), and unspecified operations performed by the physical NIC (e.g. whether a certain bit is set in a buffer descriptor). The ambiguities in the NIC specification can sometimes be solved by (i) common sense, (ii) concluding a fact from several statements in the NIC specification, (iii) making the NIC model enter a dead state, or (iv) with non-determinism or several deterministic models as discussed in the two last paragraphs in Subsection 5.1.3.2.

The NIC specification contains no information about the parallelism or atomicity of the operations of the physical NIC that are described by the NIC model. Since the transitions of the automata of the NIC model are interleaved, the operations of the physical NIC that are described by different automata are not described by the NIC model to occur in parallel. The NIC model might therefore not correctly describe the behavior of the physical NIC. This sequential description property of the NIC model is probably not an issue for two reasons. First, each transition of an automaton describes one fine-grained operation of the physical NIC, which accesses either one byte of the memory or one byte or one field of a NIC register. Second, the NIC model includes all possible interleavings of all automaton transitions (since *nic_execute* selects automaton transitions non-deterministically). If the physical NIC accesses memory or NIC registers at a granularity finer than at the byte and field granularity, the physical NIC accesses memory or NIC registers at the bit granularity, which is probably too primitive (which is motivated for memory accesses in the next Subsection 5.1.3.2).

To ease the construction of the NIC model and to minimize the number of bugs in it, the focus of the NIC model is on the operations of the physical NIC that affect which memory accesses the physical NIC performs. However, focusing on memory accesses and excluding many other operations of the physical NIC opens up the possibility of the NIC model to not describe some operations of the physical NIC that affect which memory accesses the physical NIC performs. Bugs of this type thwarts the purpose of the NIC model of being a valid model for proving that only signed Linux code is executed.

To avoid introducing bugs in the NIC model due to misunderstandings of the NIC specification, carefully selected parts of the NIC specification were read slowly several times and the NIC device driver in Linux 3.10 was studied in detail.

However, there are two exceptions with respect to the memory access focus of the NIC model. The first exception is that the NIC model describes how the physical NIC operates when a single frame is addressed by several buffer descriptors, even though Linux uses only one buffer descriptor to address a frame. Adding this description of the operation of the physical NIC to the NIC model increases the complexity of the NIC model but also its generality.

The second exception is that the NIC model also describes frame completion and teardown interrupts (the interrupts used by Linux). Describing these interrupts allows reasoning about control flow of programs executed by the physical CPU, and makes the NIC model useful for proving properties not only dependent on memory accesses. Adding descriptions of interrupts to the NIC model has negligible impact on the correctness of the NIC model for describing which memory accesses the physical NIC performs. The reason is that the interrupt logic of the NIC model is simple and decoupled from the rest of the logic of the NIC model.

Another source for introducing bugs in the NIC model is typing errors. Such and other types of bugs can potentially be found by performing correctness tests on the NIC model. Such correctness tests have not been performed since the NIC model is not implemented in HOL4. However, it is desirable to perform such tests before the NIC model is used to prove that only signed Linux code is executed. For instance, the physical NIC and the model of the NIC can be configured to access an identical set of memory addresses, and the physical memory and the model of the memory can be set to contain identical data, for several different scenarios. To test whether the NIC model correctly describes which memory accesses the physical NIC performs, it can then be checked if the physical NIC and the model of the NIC transmit and receive identical data.

Otherwise, the construction methodology of the NIC model has been simplicity to make it correct:

- Modeling the five tasks of the physical NIC as five separate and decoupled automata, thereby isolating the logic of the NIC model that describes the operations of a single task.

- Describing most transitions by one step function, most of which include a small amount of logic, making the description of one transition simple.
- Specifying the pseudocode of the NIC model to have a logical structure that is similar to the descriptions in the NIC specification of the modeled operations of the physical NIC. For instance, each if statement in the NIC model reflects one coherent decision, instead of several unrelated decisions that have been condensed into one if statement to minimize code size. This methodology keeps the logic of the NIC model simple and accurate.
- Making the NIC model enter a dead state if an operation of the physical NIC is either not clearly specified or not used by Linux, which reduces the complexity of the NIC model.

5.1.3.2 Issues regarding Memory Accesses of the NIC Model

The focus of the NIC model is to describe which memory accesses the physical NIC performs. For this purpose, the NIC model mainly describes how the CPDMA_SOFT_RESET, CPPI_RAM, HDP, CP and TEARDOWN registers affect the operation of the physical NIC. It is critical for the NIC model to describe all operations of the physical NIC that affect the content of CPPI_RAM since the content of CPPI_RAM affects which memory accesses the physical NIC performs. For instance, if two buffer descriptors overlap and the physical NIC performs a write to a field of one of the buffer descriptors, then that write can potentially also modify the buffer pointer or buffer length fields of the other buffer descriptor. Hence, the physical NIC can potentially reconfigure itself to access other physical memory addresses than originally configured by the software to access. An example of this behavior of the physical NIC is shown in Figures 31 and 32.

The NIC model describes all operations of the physical NIC that affect the content of CPPI_RAM. However, some operations of the physical NIC that perform writes to CPPI_RAM depend on values of registers of the physical NIC not included in the NIC model. The NIC model therefore describes those CPPI_RAM writes by non-deterministically selecting the value to write. Some of the buffer descriptor fields that are written non-deterministically are also related. This relationship restricts which values of those buffer descriptor fields that are valid to write. Since the values to write are chosen non-deterministically, those buffer descriptor fields might contain inconsistent values. This is not a problem because the NIC model also includes execution traces where the buffer descriptor fields are written with consistent values, since the values are chosen non-deterministically.

One important issue of the NIC model is its potential overspecification of describing the order in which certain operations are performed by the physical NIC, and the granularity of those operations (i.e. how much work each atomic operation performs). For instance, the NIC specification does not specify:

- Whether the SOP bit is set before the EOP bit in receive buffer descriptors.
- Whether all buffer descriptors are read before a received frame is stored into the data buffers addressed by those buffer descriptors.

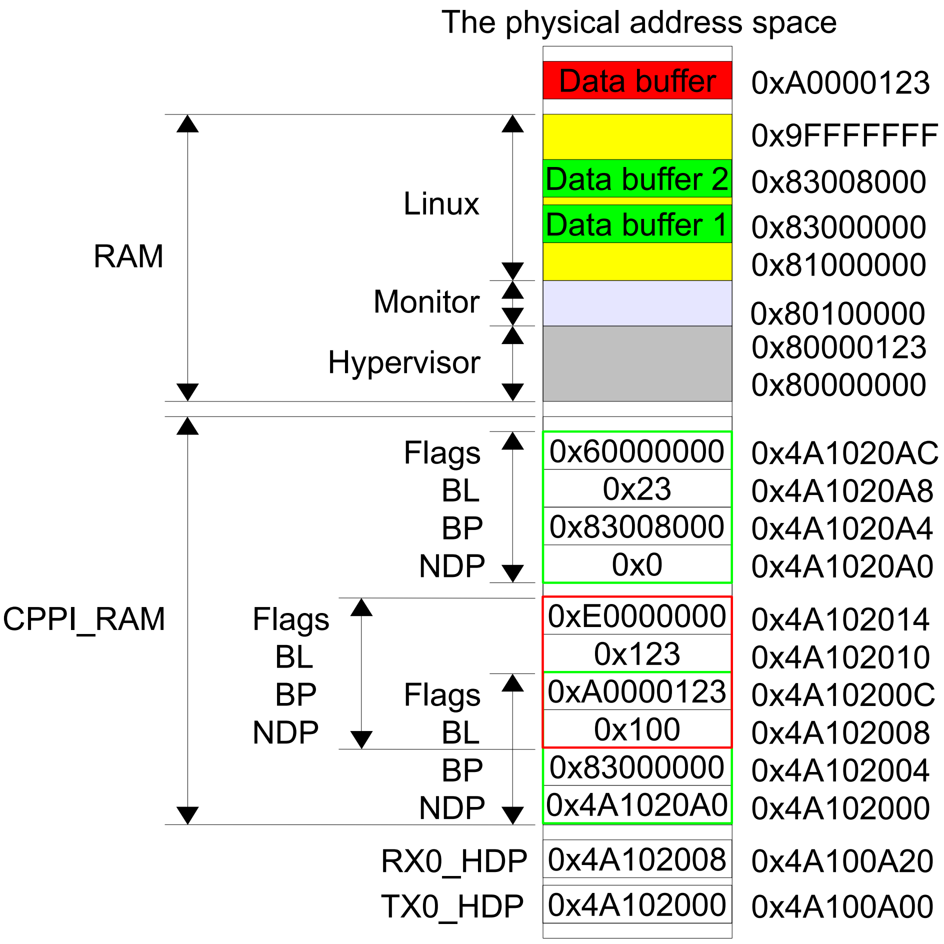


Figure 31: Overlapping buffer descriptors. The physical NIC is about to start transmission of the frame located in the two data buffers addressed by the two buffer descriptors located at 0x4A102000 (value of TX0_HDP) and 0x4A1020A0. The size of the frame is 0x123 bytes, as specified by the Buffer Length fields (BL) of these two buffer descriptors (0x100+0x23) and by the 11 least significant bits of the Flags field of the first buffer descriptor (0x123). In the Flags field of the first buffer descriptor, the start of packet and ownership bits are set to one, giving that Flags field the value 0xA0000123. The physical NIC is enabled for reception and can store the first received frame at the location addressed by the Buffer Pointer field (BP) of the buffer descriptor located at 0x4A102008 (value of RX0_HDP). Since this Buffer Pointer field overlaps the Flags field of the first buffer descriptor in the transmission DMA channel, the data buffer for reception is located at 0xA0000123. (This data buffer address happens to be outside RAM on Beaglebone Black. The NIC specification does not specify how the physical NIC operates when a data buffer is outside RAM, or when the Next Descriptor Pointer field, NDP, contains an address outside CPPI_RAM, which is the case for this receive buffer descriptor where it is equal to 0x100.) In Flags field of the receive buffer descriptor, the start of packet, end of packet and ownership bits are set to one, giving that Flags field the value 0xE0000000. The 11 least significant bits of that Flags field contains the size of a received frame, and is set by the physical NIC after a frame is received. The Buffer Length field of the receive buffer descriptor is set to 0x123, indicating that the size of the addressed data buffer is 0x123 bytes.

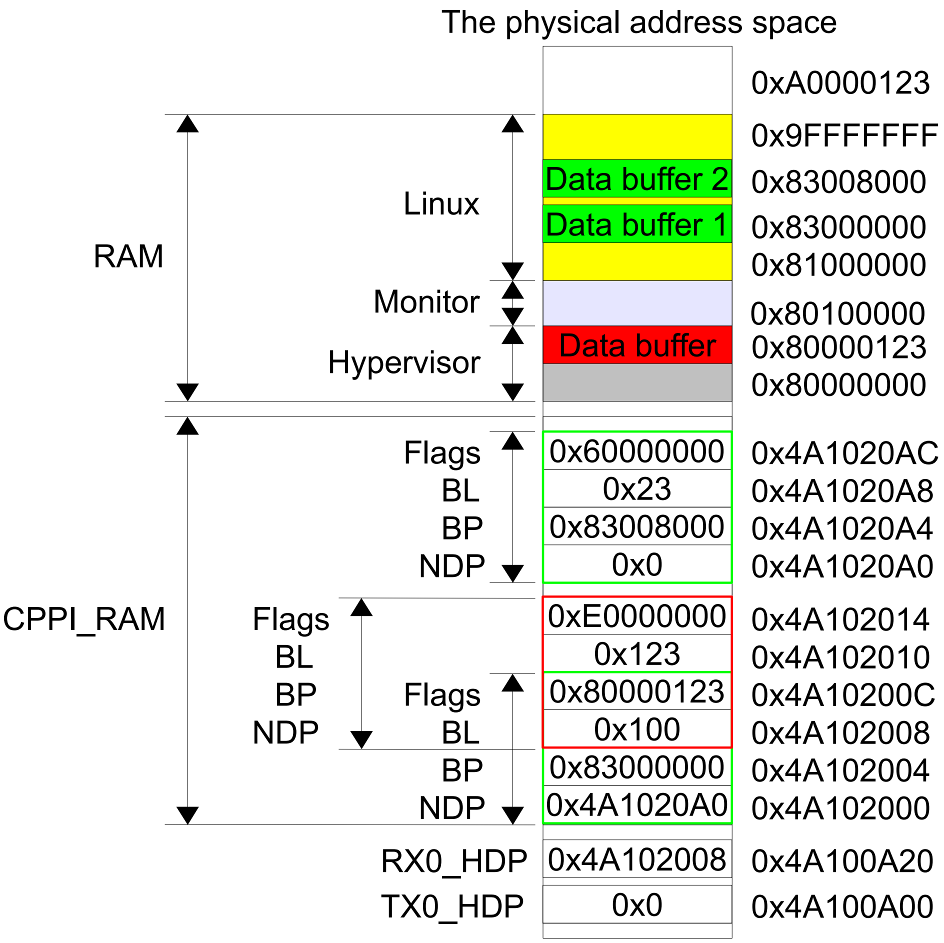


Figure 32: Modifications of overlapping buffer descriptors potentially giving the control of the system to Linux. The figure shows the same state as is shown in Figure 31 but after the physical NIC has processed the two buffer descriptors in the transmission queue. (No frame has been received during this processing.) When the physical NIC has transmitted a frame, it clears the ownership bit in the buffer descriptor that has the start of packet bit set (bit 29 in the Flags field). In this case, the physical NIC has cleared bit 29 of the Flags field of the buffer descriptor located at 0x4A102000, changing its value from 0xA0000123 to 0x80000123. This makes the overlapping Buffer Pointer field of the receive buffer descriptor at 0x4A102008 contain 0x80000123. Hence, if the physical NIC now receives a frame, it will store that frame at 0x80000123. Since the Buffer Length field of the receive buffer descriptor is equal to 0x123, only the first 0x123 bytes of that frame will be stored in RAM, possibly ending at 0x80000245. If that memory region is allocated to the hypervisor (as is shown in this example), the physical NIC might overwrite code or data of the hypervisor. This might result in that the control of the system is given to Linux.

- The order in which bytes are read of buffer descriptors or frames to transmit.

- The order in which the operations listed in the previous three item bullets are performed and the granularity of these operations.

Due to the lack of this sort of information, assumptions have been made about the order in which certain operations are performed by the physical NIC and the granularity of those operations.

The lack of information of the granularity of the operations is handled by specifying the NIC model to describe those operations at a granularity that is as fined-grained as reasonable. That is, according to the NIC model, an atomic operation of the physical NIC is accessing at least one byte of the memory or one byte or field of a NIC register. As discussed in the previous subsection, the granularity of NIC register accesses performed by the physical NIC is probably not more fine-grained than that.

Consider the granularity of memory accesses. The physical CPU, memory and NIC communicate (excluding interrupts) by means of the OCP 2.2 protocol [32]. The OCP 2.2 protocol allows data transfers that are not byte multiples: "OCP supports word sizes [transfer width in bits] of power-of-two and non-power-of-two as would be needed for a 12-bit DSP core" [87]. Data transfers not being byte multiples are probably not implemented in the chip containing the physical CPU, memory controller and NIC. The reason is that it does not make sense for these components to intact at such a granularity: The ARMv7 CPU has only instructions for 8-bit, 16-bit, 32-bit and 64-bit memory accesses, ARM instructions are 32 bits wide (which are fetched from memory) [33], and the physical NIC only transfers frames whose sizes are multiples of bytes [32].

However, the potential overspecification of the order of certain operations might be a problem. If sequences of operations that perform writes to CPPI_RAM are not described accurately by the NIC model, the NIC model might not read some values of CPPI_RAM that the physical NIC reads. Since the content of CPPI_RAM determines which memory accesses the physical NIC performs, the NIC model might therefore not describe certain memory accesses that the physical NIC performs. This is of course not desirable since memory accesses have a critical role in the reliability of the proof of that only signed Linux code is executed. There are at least two potential solutions to this potential overspecification problem.

One potential solution is to construct a single NIC model which non-deterministically selects the next transition to perform, thereby describing several orders to perform a certain set of operations of the physical NIC. Another potential solution is to construct multiple NIC models where each model is deterministic with respect to the order of the operations, but such that all NIC models together cover all possible orderings of performing the operations. A property is then proved on the single non-deterministic NIC model or on all deterministic NIC models.

The reason for having multiple deterministic NIC models instead of one non-deterministic NIC model is because a set of deterministic NIC models probably describes the operation of the physical NIC more accurately. The next operation performed by the physical NIC is probably not selected non-deterministically. The increased accuracy provided by a set of deterministic NIC models compared to one non-deterministic NIC model is needed if properties to prove depend on whether operations of the physical are selected deterministically or non-deterministically.

Having these two deterministic and non-deterministic approaches in mind for describing the operation of the physical NIC, it might be desirable to simplify the current NIC model. The only relevant simplification is to modify the current NIC model to only describe how the physical NIC operates when a single frame is only addressed by a single buffer descriptor, instead of being addressed by several buffer descriptors. Such a simplified NIC model can probably be used as a starting point to make it feasible to describe the operation of the physical NIC according to these two deterministic and non-deterministic approaches.

5.1.3.3 Conclusions

To summarize, the following questions are related to the problems that exist with the NIC specification and the NIC model:

- What do certain operations actually perform (e.g. what does the reset operation actually do)?
- What is the granularity of certain operations (e.g. is the CP register set to 0xFFFFFFFC simultaneously as the HDP register is zeroed or are these two operations performed by two separate atomic operations)?
- What is the order of performance of certain operations (e.g. is the SOP bit set first and then the EOP bit or are these two operations performed in the opposite order)?

Assuming that the NIC specification mentions the operations of the physical NIC that affect which memory accesses the physical NIC performs (which is a reasonable assumption), and since the transitions of the NIC model are relatively fine-grained, the critical issue with the NIC model is its description of the order in which the physical NIC performs its operations.

The NIC model, in its current state, might not be perfect but it is still a well-made ground that can be used for making relatively accurate analyzes of which memory accesses the physical NIC performs. The NIC model provides formal descriptions for the interesting behavior of the physical NIC (performed memory accesses and asserted interrupts as the physical NIC is configured by the NIC device driver in Linux 3.10), and it also illustrates how the operation of the physical NIC can be described formally. The applicability of the NIC model also depends on the desired reliability of the formal proof of that only signed Linux code is executed.

5.2 Real Model

The real model is a transition system that describes the operation of the hardware upon which the hypervisor, the monitor and Linux run on. Subsection 5.2.1 briefly describes the real model. Subsection 5.2.2 discusses the accuracy of the real model. Subsection 5.2.3 discusses some issues and the applicability of the real model. Subsection 5.2.4 presents the formalization of the real model, and which is used in the proof plan in Chapter 6 to reason that only signed Linux code is executed.

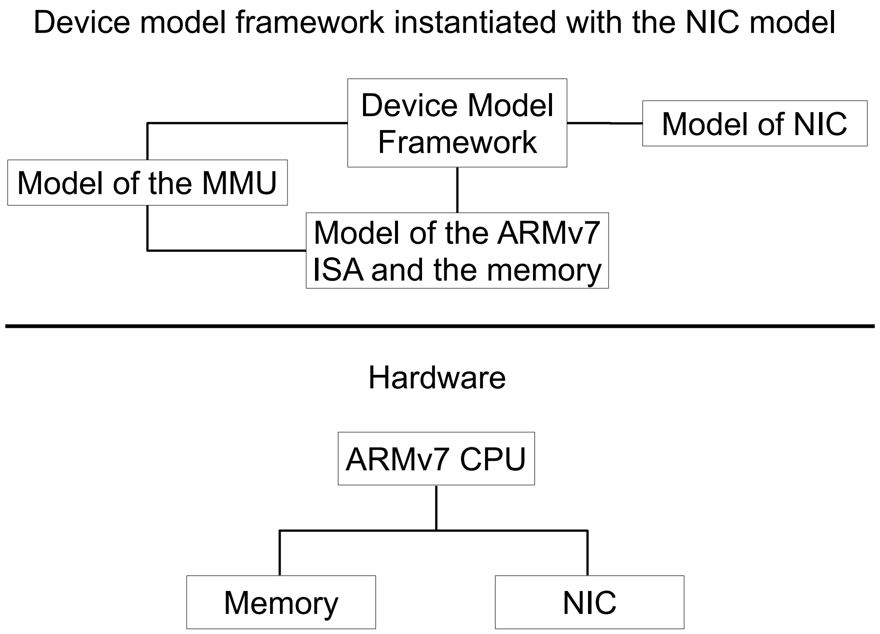

Device model framework instantiated with the NIC model
Device Model Framework
Model of NIC
Model of the MMU
Model of the ARMv7 ISA and the memory
Hardware
ARMv7 CPU
Memory
NIC

Figure 33: The device model framework instantiated with a HOL4 implementation of the NIC model. The resulting hardware model describes system executions of hardware consisting of an ARMv7 CPU with an MMU, a memory, and the NIC on BeagleBone Black. The real model is obtained by instantiating the memory state component of this hardware model with the binary images of the hypervisor, the monitor and Linux. The real model allows formal reasoning of how this hardware executes the hypervisor, the monitor and Linux, and the real model can therefore be used to formally prove that only signed Linux code is executed in this system.

5.2.1 Device Model Framework Instantiated with the NIC Model

By instantiating the device model framework (described in Subsection 2.4.1) with a HOL4 implementation of the NIC model and the memory state component with the binary images of the hypervisor, the monitor and Linux, a formal description is obtained of all executions of a system consisting of an ARMv7 CPU, a memory, the NIC on Beaglebone Black, the hypervisor, the monitor and Linux. This formal description is in the form of a transition system, called the real model. Figure 33 illustrates the structure of the device model framework instantiated with a HOL4 implementation of the NIC model (cf. Figure 13 in Subsection 2.4.1). The state of the real model consists of the state of the device model framework but with the general device state component instantiated with the state of the NIC model. That is, the state of the real model consists of the states of the ARMv7 ISA model (including the state of the memory), the NIC model, and the device model framework. (The MMU model does not involve states as will be seen in Subsection 5.2.4.3.)

There are three types of transitions depending on which state components of a real model state a transition might affect:

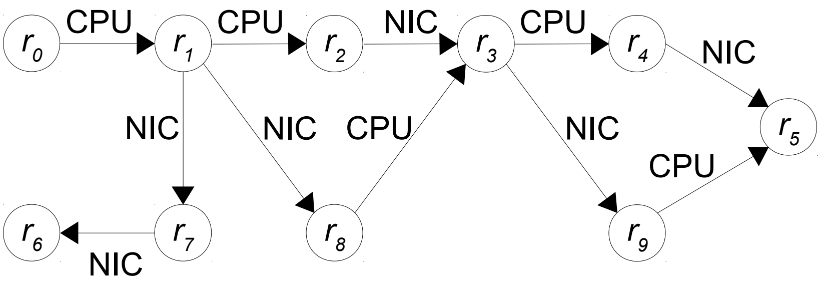


Figure 34: A graphical illustration of transitions in the real model. A transition with the label CPU describes the execution of one CPU instruction. A transition with the label NIC describes one execution step performed by the physical NIC. These latter transitions are described by nic_execute and memory_byte (see Subsection 5.1.1). The device model framework function advance_single applies nic_execute to let the NIC model perform an autonomous transition and potentially issue a memory read request. If a memory read request is issued, advance_single also applies memory_byte to give the value of the addressed memory byte to the NIC model (see Subsection 2.4.1). This figure illustrates the non-determinism in both the device model framework and in the NIC model. The transitions $r_1 \rightarrow r_2$ and $r_1 \rightarrow r_8$ are a result of the non-determinism in the device model framework. In the real model state r_1, if the non-deterministic scheduler of the device model framework selects the CPU model to perform a transition, the transition $r_1 \rightarrow r_2$ is performed. If the NIC model is selected, the transition $r_1 \rightarrow r_8$ is performed. The transitions $r_1 \rightarrow r_7$ and $r_1 \rightarrow r_8$ are a result of the non-determinism in the NIC model, depending on which automaton of the NIC model that is selected by nic_execute to perform the next transition of the NIC model.

- Transitions describing the execution of CPU instructions not accessing NIC registers: Each such transition only affects the states of the models of the physical CPU and the physical memory.
- Transitions describing the execution of the physical NIC: Each such transition only affects the states of the models of the physical NIC and the physical memory.
- Transitions describing the execution of CPU instructions accessing NIC registers: Each such transition only affects the states of the models of the physical CPU and the physical NIC.

The non-deterministic scheduler of the device model framework allows the real model to include all possible interleavings of these transitions of the CPU and NIC models. Figure 34 gives a graphical illustration of transitions in the real model (cf. Figure 14 in Subsection 2.4.1).

For these reasons, the real model can be used to formally reason about the hardware that executes the hypervisor, the monitor and Linux. In particular, the real model can be used to formally prove in HOL4 that in this system, only signed Linux code is executed. The reason for using the device model framework as a base for formally proving that only signed Linux code is executed is because

PROSPER is familiar with the CPU model and the device model framework, and the usage of the device model framework therefore makes the results of this thesis project usable for PROSPER.

An important aspect of the real model is the four I/O device model functions that the device model framework applies for a device: *d_read*, *d_write*, *progress* and *receive* (described in Subsection 2.4.1). These four functions are instantiated by the NIC model functions *read_nic_register*, *write_nic_register*, *nic_execute* and *memory_byte*, respectively (described in Subsection 5.1.1). The device model framework should apply these four NIC model functions in the following situations. *read_nic_register* and *write_nic_register* are applied when the CPU model performs a transition that accesses a physical address in the interval [0x4A100000, 0x4A103FFF] (the physical addresses of the NIC registers). *nic_execute* is applied by *advance_single* when the non-deterministic scheduler of the device model framework decides that the NIC model shall perform an autonomous transition. If *nic_execute* returns a memory read request, *advance_single* also applies *memory_byte* to give the NIC model the value of the requested byte.

5.2.2 Accuracy of the Real Model

This subsection discusses how accurately the real model describes the hardware by considering:

- Interactions between the physical CPU, memory and NIC.
- Cache behavior.
- Some assumptions made in the real model about the hardware.
- Accuracy and correctness of the individual models constituting the real model.

5.2.2.1 Interaction between the Physical CPU, Memory and NIC

In reality the physical CPU and the physical NIC execute in parallel, but in the real model their executions are described to occur in serial. The parallel behavior of the hardware might still not be excluded by the real model. The reason is that when the physical CPU and the physical NIC access different resources, they execute identically irrespectively of whether they execute in parallel or in serial. The critical behavior for the real model to describe correctly is therefore the interaction between the physical CPU and the physical NIC. That interaction occurs via interrupts, memory accesses and NIC register accesses. This subsection compares how the real model describes these three types of interactions with how the hardware performs them.

In the real model NIC interrupts are generated and detected atomically. This description is probably correct since it is enough for the hardware to communicate interrupts with a single bit.

Consider memory accesses performed by the CPU model. The CPU model atomically accesses 8-bit bytes that are byte aligned, 16-bit halfwords that are halfword aligned and 32-bit words that are word aligned in memory. Memory accesses performed by the CPU model that are unaligned are split up into several atomic bytewise accesses. This description of memory accesses is in accordance with the ARMv7 specification [33] and the operation of the memory controller that completes each memory access before it starts to process another one [32].

Consider memory accesses performed by the NIC model. The NIC model accesses single bytes in memory atomically. Due to the non-deterministic scheduler of the device model framework, the real model includes sequences of consecutive transitions of the NIC model, where each such transition accesses a single byte in memory. Hence, if the physical NIC accesses multiple bytes in memory atomically, that behavior is also described by the real model. As was motivated in Subsection 5.1.3.2, each atomic memory access performed by the physical NIC probably only transfers a multiple of bytes. It is therefore probable that the real model correctly describes memory accesses performed by the physical NIC.

Consider NIC register accesses. In the real model, if the CPU model accesses a NIC register and the address is aligned, the access is atomic. If the address is unaligned, the NIC model enters a dead state, symbolizing an unknown operation. According to the ARMv7 specification, aligned accesses are atomic. Considering the meaning of the physical NIC registers as described by the NIC specification, it does not make sense for the physical NIC to not atomically return all 32 accessed bits of a NIC register. For these reasons the real model probably correctly describes how the physical CPU accesses NIC registers. The NIC model describes NIC register accesses at the byte and field granularity, and as discussed in Subsection 5.1.3.1 it is unlikely that the physical NIC accesses NIC registers at the bit granularity.

Since the real model probably describes interrupts, memory accesses and NIC register accesses correctly, it is probably not a problem that the real model describes the executions of the physical CPU and NIC to occur in serial, despite that the executions occur in parallel in reality.

5.2.2.2 Lack of Cache Behavior

Cache behavior is also relevant to consider. The property of that only signed Linux code is executed depends on the CPU's view of memory content. That is, both cache and memory contents. It is therefore desirable that the real model correctly reflects the contents of the caches and the memory. However, the real model does not describe cache behavior.

Imagine that the hardware performs the following steps:

1. The physical NIC writes a byte to memory at a location that is also currently being held in the cache.
2. The physical CPU reads a 32-bit word from memory that is transferred to the cache. None of the bytes of this word are located at the address where the byte written by the physical NIC in step 1 is stored. The transfer of the read 32-bit word to the cache causes the cache to evict the cache line that addresses the byte that the physical NIC wrote in step 1. This causes the byte the physical NIC wrote in step 1 to be overwritten with the byte value that is stored in the cache of the evicted cache line.

3. The physical CPU reads a 32-bit word from memory that is transferred to the cache. One of the constituent bytes of this word is located at the address where the byte written by the physical NIC in step 1 is stored. This read causes the cache line that was evicted in step 2 to be transferred from memory back to the cache. Since the value of the byte that the physical NIC wrote in step 1 was overwritten in step 2, the physical CPU reads the value that the byte had before step 1 occurred. The physical CPU does therefore not read the value that the physical NIC wrote.

The real model describes these three steps as follows:

1. The NIC model writes the byte value to memory.
2. The CPU model reads a 32-bit word from memory that does not share an address with the byte the NIC model wrote in step 1.
3. The CPU model reads the 32-bit word from memory that contains the byte value the NIC model wrote in step 1.

In this scenario the CPU model reads one value while the physical CPU reads another value. This might not be a problem if the non-deterministic scheduler of the device model framework can produce another interleaving that reflects the behavior of the hardware. In this case the scheduler can produce the interleaving where the CPU model performs its two reads before the NIC model performs its write. Such a scheduling makes the CPU model read the same value as the physical CPU. A detailed analysis is required to answer whether for each execution trace that can be generated by the hardware, there exists an execution trace in the real model, such that the CPU model observes the same values in the execution trace in the real model that the physical CPU observes in the execution trace generated by the hardware. Hence, the lack of description of cache behavior in the real model might be an issue.

5.2.2.3 Assumptions in the Real Model

This subsection discusses whether two assumptions that the real model does about the hardware affect its accuracy with respect to describing hardware behaviors that are relevant for proving that only signed Linux code is executed. The first assumption is that devices do not react to the physical CPU reading their registers such that those reactions affect the read register values. Since reads of the physical NIC registers that are included in the NIC model do not cause side effects, this assumption is not a problem.

The second assumption is that the hardware immediately satisfies memory read requests issued by the physical NIC. This means that in the real model no transitions of the CPU or NIC models occur in between a pair of transitions of the NIC model that issue a memory read request and handle the corresponding reply. When the hardware satisfies memory read requests issued by the physical NIC is unspecified. Hence, the hardware might satisfy memory read requests issued by the physical NIC at arbitrary points in time. This raises the concern of whether this limitation of the real model causes it to omit descriptions of hardware behaviors that are relevant for proving that only signed Linux code is executed.

To answer whether it is a problem that the real model immediately satisfies memory read requests issued by the NIC model, two execution traces generated by the CPU and NIC models are compared. Consider a trace of a general form, which starts with a memory read request transition, followed by an arbitrary sequence of transitions of the CPU and NIC models, and ends with a corresponding memory read request reply transition. Consider also a trace in the real model that starts from the same state as the general trace starts from. The first transition is the same memory read request transition that is the first transition in the general trace. The second transition is the corresponding memory read request reply transition. The following sequence of transitions is generated by applying the transition functions of the CPU and NIC models in the same order as they were applied to generate the sequence of transitions in the general trace, excluding the memory read request and reply transitions. (Two such traces are shown in Figure 35.) These two traces describe identical hardware behaviors, except for when the issued memory read request is satisfied. The reasons for why the two traces describe nearly identical hardware behaviors are explained in what follows.

Memory read request reply transitions are independent of the following transitions, and vice versa:

- Transitions performed by the CPU model: The operations performed by memory read request reply transitions do not operate on the byte values read from the memory (the physical NIC ignores memory content). They only operate on and update state components that are only accessed by the transitions performed by the transmission automaton of the NIC model. Memory read request reply transitions are therefore independent of transitions performed by the CPU model, and vice versa.
- Transitions performed by the NIC model in between a pair of memory read request and reply transitions: Consider the following three properties of the NIC model. First, memory read request reply transitions are only performed by the transmission automaton. Second, the transmission automaton waits for a pending memory read request to be satisfied before it performs additional transitions (the transmission automaton is active during this time). Third, the initialization and transmission teardown automata do not perform transitions while the transmission automaton is active. These three properties imply that only the reception and reception teardown automata can perform transitions of the NIC model in between a pair of memory read request and reply transitions. Memory read request reply transitions are independent of transitions performed by the reception and reception teardown automata, and vice versa, for the same reason as memory read request reply transitions and transitions performed by the CPU model are independent of each other.

Hence, a memory read request reply transition is independent of the transitions that occur before it and after its corresponding memory read request transition, and vice versa.

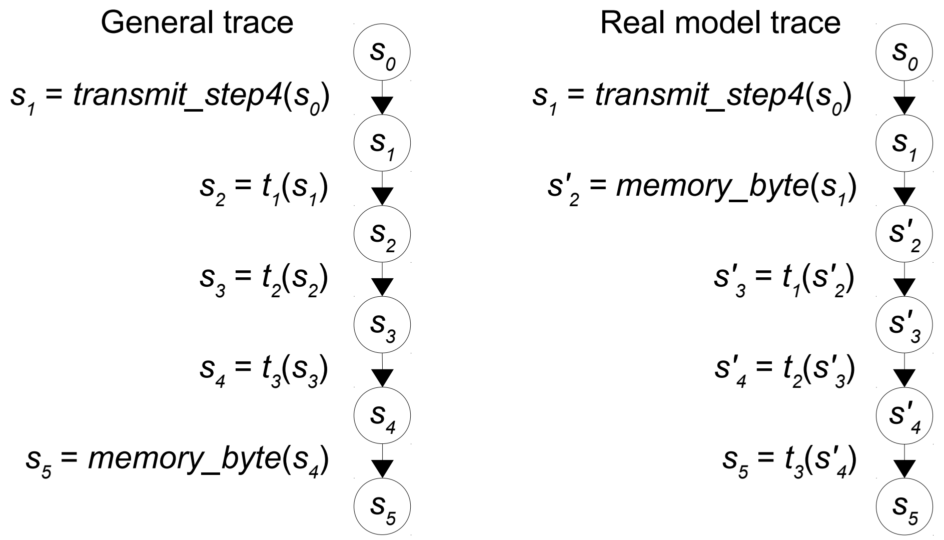


Figure 35: An example of two traces in which memory read requests issued by the NIC model are satisfied at different points in time. Only step function four of the NIC model, transmit_step4, issues memory read requests. The NIC model function memory_byte handles memory read request replies. Each function t_i, $1 \leq i \leq 3$, is either the transition function of the CPU model, possibly interacting with the two NIC model functions describing NIC register accesses (read_nic_register and write_nic_register) or the transition function of the NIC model describing execution steps of the physical NIC (nic_execute). (These functions are defined to operate on CPU and NIC model states, but for simplicity they are shown as operating on real model states. In this figure, the intension is that the operations performed by these functions are the operations they are defined to perform, but performed on the state components of real model states that hold CPU and NIC model states.) The trace to the left has a general form since the memory read request is not satisfied immediately. The trace to the right is included in the real model since the memory read request is satisfied immediately. The traces describe the same hardware behavior, with the exception of when the memory read request is satisfied. There are three reasons for this similarity. First, the operations performed by memory_byte do not the operate on the byte value read from memory. Second, the state components operated on or updated by memory_byte are not operated on or updated by t_i, $1 \leq i \leq 3$, and vice versa. Third, the traces are generated by the same sequence of function applications, excluding the application of memory_byte. Due to the independence between memory_byte and t_i, $1 \leq i \leq 3$, it does not matter if t_i, $1 \leq i \leq 3$, are applied in the same order before or after memory_byte is applied: The operations performed by the application of memory_byte on s_4 and the operations performed by the application of memory_byte on s_1 are identical and those operations operate on and update the same state components with equal values, and similarly for the operations performed by the applications of t_i on s_i and s'_{i+1}, $1 \leq i \leq 3$. Hence, there exists a one-to-one relationship between the transitions in the general trace and the transitions in the real model trace: Related transitions perform identical operations. The transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, $s_2 \rightarrow s_3$, $s_3 \rightarrow s_4$ and $s_4 \rightarrow s_5$ in the general trace are related to the transitions $s_0 \rightarrow s_1$, $s'_2 \rightarrow s'_3$, $s'_3 \rightarrow s'_4$, $s'_4 \rightarrow s_5$ and $s_1 \rightarrow s'_2$ in the real model trace, respectively.

The general trace and the real model trace start from the same state. Also, their sequences of transitions are generated by the same sequence of function applications, except for the generations of the memory read request reply transitions, which are independent of the transitions following the memory read request transitions and vice versa. There exists therefore a one-to-one relationship between the transitions in the general trace and the transitions in the real model trace: For each pair of related transitions, those two transitions perform identical operations operating on and updating the same state components with equal values. In addition, the transitions in a related pair occur at the same positions within their traces, excluding the memory read request reply transitions. Hence, the only difference between the general trace and the real model trace is when the memory read request is satisfied. Figure 35 and its caption clarify this reasoning.

For proving that only signed Linux code is executed, the relevant hardware property that the real model must describe accurately is how the physical CPU executes CPU instructions. Traces of the general form describe hardware behavior where memory read requests are satisfied at arbitrary points in time. The only difference between general traces and real model traces is their description of when memory read requests are satisfied. This only difference means that general traces and real model traces identically describe how the physical CPU executes CPU instructions. Describing memory read requests to be satisfied immediately therefore does not cause the real model to omit descriptions of hardware behaviors that are relevant for proving that only signed Linux code is executed. Hence, the conclusion is that it is not a problem that the real model assumes that the hardware immediately satisfies memory read requests issued by the physical NIC.

5.2.2.4 Accuracy of Individual Models

It is also important to consider potential bugs in the models that constitute the real model. The ARMv7 ISA model [84] has been tested by means of large test programs, consisting of randomly chosen instructions in order to cover the complete model. These test programs were executed on the model and on three development boards with their execution results compared. Issues of the NIC model were discussed in the previous chapter, and how the correctness of the MMU model [86] has been checked is unknown.

5.2.3 Conclusion and Discussion

The most important issues of the real model are (with respect to prove that only signed Linux code is executed):

- The potential overspecification of the order of the operations of the physical NIC.
- The absence of describing the cache behavior of the hardware.

Even if the real model does not describe the complete behavior of the hardware, it still describes most of the behavior of the hardware that is relevant for proving that only signed Linux code is executed. The real model provides this description by means of a transition system. Each transition in the transition system describes how the execution of one CPU instruction or one fine-grained NIC operation modifies a detailed hardware state. By means of non-deterministic schedulers, the transition system includes all possible interleavings of these transitions. These features of the real model, combined with that most of the real model is already implemented in a theorem prover (namely HOL4), makes the real model attractive to use to formally prove that only signed Linux code is executed.

5.2.4 Formalization of the Real Model

The purpose of this section is to formally describe the real model on paper. This formal description is what the proof plan in Chapter 6 refers to when reasoning that only signed Linux code is executed. Subsection 5.2.4.1 describes the data type of the states in the real model. Subsection 5.2.4.2 describes the transition rules that describe the transitions in real model. Subsection 5.2.4.3 uses the data type of the states and the transition rules to define a four tuple that denotes the sets of states, initial states, transitions and execution traces in the real model. The components of that four tuple are used in the proof plan to refer to the real model.

5.2.4.1 States in the Real Model

A state in the real model has the data type real_state, an instance of which is called a real state. A real state represents the state of the hardware at particular point in a hardware system execution. The state components of a real state are the states of the ARMv7 ISA model, which includes the state of the memory model, the NIC model, and the device model framework. Figure 36 shows the state components of real_state.

5.2.4.2 Transition Rules of the Real Model

The transition rules described in this subsection describe all transitions in the real model (see Subsection 2.1.2 for an explanation of transition rules and labeled transition systems):

- CPU transitions: Generated by the transition rules concerned with the CPU model. One CPU transition in the real model describes the execution of one CPU instruction.
- NIC transitions: Generated by the transition rules concerned with the NIC model. One NIC transition in the real model describes the execution of one fine-grained operation of the physical NIC. If that operation issues a memory read request, a NIC transition also describes one fine-grained operation of the physical NIC for handling the reply to that memory read request. That is, one NIC transition in the real model is one autonomous transition of the NIC model possibly followed by one memory read request reply transition of the NIC model, if the autonomous transition issued a memory read request.

The transition rule that generate CPU transitions are described first and then the transition rules that generate NIC transitions. The transition rules reflect how the device model framework is implemented (similarly to how the device model framework functions *next* and *advance_single* operate, as described in Subsection

```
real_state = (
    cpu_state cpu,
    word32 → word8 memory,
    nic_state nic
)
cpu_state = (
    user_regs uregs,
    privileged_regs pregs,
    system_regs sregs,
    cp15_regs cp15,
    bool int
)
user_regs = (
    word32 r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15
)
privileged_regs = (
    word32 r8_fiq, r9_fiq, r10_fiq, r11_fiq, r12_fiq,
    word32 r13_svc, r13_abt, r13_und, r13_irq, r13_fiq,
    word32 r14_svc, r14_abt, r14_und, r14_irq, r14_fiq
)
system_regs = (
    word32 CPSR,
    word32 SPSR_svc, SPSR_abt, SPSR_und, SPSR_irq, SPSR_fiq
)
cp15_regs = (
    word32 TTBR0, DACR, DFAR
)
```

Figure 36: The data type of a real state, denoted by real_state. This syntax follows the pseudocode notation described in Appendix A. real_state contains three state components, cpu, memory and nic, having data types cpu_state, word32 → word8 and nic_state, respectively. cpu and memory contain the state of the CPU model and nic contains the state of the NIC model. S_{real} denotes the set of all possible instances of real_state. If $r \in S_{real}$, then the cpu, memory and nic state components of r are referred to as r.cpu, r.memory, and r.nic, respectively. The cpu state component has in turn five state components (only the five state components of cpu that are relevant for this thesis are shown). cpu.uregs records the contents of all general-purpose registers that are accessible in non-privileged mode. cpu.pregs records the contents of all general-purpose registers that are only accessible in privileged mode. cpu.sregs records the contents of all program status registers. cpu.cp15 records the contents of the three coprocessor 15 registers, which are related to the MMU. (See Subsection 2.3.1 for an explanation of the meaning of these four state components.) cpu.int records whether the NIC model asserts an interrupt or not. The memory state component is a function that records the contents of the 512 MB of RAM on BeagleBone Black. It takes a 32-bit physical address as argument and returns the byte value stored at that address. This function is unused for the other 3.5 GB of the physical address space. The nic state component is explained by comments in the definition of nic_state in Section C.3.

2.4.1), but not how the CPU and NIC models are implemented. However, the transition rules are not completely accurate. In particular, CPU instructions performing reads or writes to multiple locations in the physical address space are not described. This is not an issue for the purposes of the proof plan (describing how it can be formally proved that only signed Linux code is executed).

5.2.4.2.1 Transition Rules for CPU Transitions

The ARMv7 ISA and MMU models [84, 86] can be considered to together constitute a CPU model that describes how the physical CPU executes individual CPU instructions according to the ARMv7 specification [33]. The transition function of that CPU model is denoted by *cpu_execute*:

$$(\text{cpu\_state}, \text{word32} \rightarrow \text{word8})\ \mathit{cpu\_execute}(\text{cpu\_state}\ \mathit{cpu}, \text{word32} \rightarrow \text{word8}\ \mathit{memory}).$$

The arguments *cpu* and *memory* of *cpu_execute* describe a physical CPU state and a physical memory state, respectively. The return values of *cpu_execute* describe the physical CPU and memory states that the physical CPU and memory enter after the physical CPU has executed one CPU instruction from the physical CPU and memory states described by *cpu* and *memory*.

In what follows, it is necessary to find which execution mode the CPU model is in. The function *mode* is used for this purpose:

$$\{\mathit{usr}, \mathit{fiq}, \mathit{irq}, \mathit{svc}, \mathit{abt}, \mathit{und}, \mathit{sys}, \perp\}\ \mathit{mode}(\text{real\_state}\ r).$$

If the execution mode of the CPU model is defined in the real state *r*, then *mode* returns the corresponding identifier. Otherwise ⊥ is returned. (*mode* depends only on *r.cpu.sregs.CPSR*[4:0].)

There are three CPU transition rules, depending on whether the next CPU instruction execution does not access a NIC register, reads a NIC register, or writes a NIC register. The three CPU transition rules correspond to the first step of the function *next* of the device model framework, as described in Subsection 2.4.1.

The transition rule for CPU instruction executions not accessing a NIC register:

$$\frac{(\mathit{cpu}', \mathit{memory}') = \mathit{cpu\_execute}(\mathit{cpu}, \mathit{memory}) \wedge \neg \mathit{nic\_access}(\mathit{cpu}, \mathit{memory})}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}) \rightarrow_{\mathit{cpu\_exec\_type}(\mathit{cpu}, \mathit{cpu}')} (\mathit{cpu}', \mathit{memory}', \mathit{nic})}.$$

This transition rule has the following meaning. Assume the following:

- The next instruction execution by the CPU model transforms *cpu* and *memory* to *cpu'* and *memory'*, respectively.
- The instruction execution did not access a NIC register, indicated by the predicate *nic_access*.

There is then a transition from the real state (*cpu*, *memory*, *nic*) to the real state (*cpu'*, *memory'*, *nic*). The label of that transition is determined by the function *cpu_exec_type*:

$$\{\mathit{CPU}, \mathit{EXC}, \mathit{RET}\}\ \mathit{cpu\_exec\_type}(\text{cpu\_state}\ \mathit{cpu}, \text{cpu\_state}\ \mathit{cpu}').$$

cpu_exec_type is defined as follows:

- If $mode(cpu) = usr \wedge mode(cpu') \in \{fiq, irq, svc, abt, und\}$:

  $$cpu\_exec\_type(cpu, cpu') = EXC.$$

  That is, the execution of the next instruction raises an exception (sys mode cannot be entered from usr mode).

- If $mode(cpu) \in \{fiq, irq, svc, abt, und, sys\} \wedge mode(cpu') = usr$:

  $$cpu\_exec\_type(cpu, cpu') = RET.$$

  That is, the execution of the next instruction causes the CPU model to return from an exception handler.

- Otherwise:

  $$cpu\_exec\_type(cpu, cpu') = CPU.$$

  That is, the CPU model does not raise an exception nor returns from an exception handler when executing the next instruction.

The transition rule for NIC register reads is:

$$\frac{\begin{array}{c} pa = cpu\_read\_nic\_register(cpu, memory) \wedge \\ (nic', val) = read\_nic\_register(nic, pa) \wedge \\ memory' = memory[pa \mapsto val[7{:}0], \\ pa + 1 \mapsto val[15{:}8], pa + 2 \mapsto val[23{:}16], pa + 3 \mapsto val[31{:}24]] \wedge \\ (cpu', memory'') = cpu\_execute(cpu, memory') \end{array}}{(cpu, memory, nic) \rightarrow_{CPU} (cpu', memory, nic')}.$$

The meaning of this transition rule is as follows. Assume the following:

- From the CPU and memory model states, *cpu* and *model*, respectively, the next CPU instruction execution reads the NIC register at physical address *pa*, as indicated by the function *cpu_read_nic_register*. If a NIC register is not accessed, *cpu_read_nic_register* returns $\perp$.
- The NIC model function describing NIC register reads, *read_nic_register*, updates the NIC model state *nic* to *nic'*, and returns the value *val* as the read value of the NIC register located at physical address *pa* in the NIC model state *nic*.
- *memory'* is equal to *memory* but *memory'* maps the arguments *pa*, *pa* + 1, *pa* + 2 and *pa* + 3 to the byte values *val*[7:0], *val*[15:8], *val*[23:16] and *val*[31:24], respectively. That is, *memory'*(*pa*) = *val*[7:0], etc.
- The next CPU instruction execution (which reads a location in the physical address space as determined by *cpu_read_nic_register*) causes the CPU model to update the CPU and memory model states from *cpu* and *memory'* to *cpu'* and *memory''*, respectively.

  Since the CPU model only operates on the CPU and memory model states, some mechanism is needed to make the CPU model operate on the current NIC register value. In this case, the mechanism is to set the entries of *memory'* where the read NIC register is located to the read NIC register value *val*. This causes the CPU model to operate on the value of the NIC register located at *pa*. (The state of the memory model is used as if it represented the physical address space and not the physical memory.) In the real model, this interaction between the CPU model and the NIC model is handled by the device model framework.

Under these four assumptions, there exists a transition from (*cpu*, *memory*, *nic*) to (*cpu'*, *memory*, *nic'*) with the label *CPU*.

The transition rule for NIC register writes is:

$$\frac{\begin{array}{c} pa = cpu\_write\_nic\_register(cpu, memory) \wedge \\ (cpu', memory') = cpu\_execute(cpu, memory) \wedge \\ val = memory'(pa + 3) :: memory'(pa + 2) :: memory'(pa + 1) :: memory'(pa) \wedge \\ nic' = write\_nic\_register(nic, pa, val) \end{array}}{(cpu, memory, nic) \rightarrow_{CPU} (cpu', memory, nic')}.$$

The meaning of this transition rule is as follows. Assume the following:

- The next instruction execution of the CPU model writes the NIC register with the physical address *pa*, as indicated by *cpu_write_nic_register*. *cpu_write_nic_register* is similar to *cpu_read_nic_register* but returns a physical address of a NIC register if and only if the next instruction execution of the CPU model writes that NIC register. Otherwise, ⊥ is returned.
- The next instruction execution of the CPU model updates the CPU and memory model states from *cpu* and *memory* to *cpu'* and *memory'*, respectively.
- Since *cpu_execute* only operates on the CPU and memory model states, *cpu_execute* updates the memory state to contain the value *val* that is intended to be written to the addressed NIC register. The symbol '::' denotes the concatenation operator, which is used to concatenate the four bytes the CPU model intended to write to the addressed NIC register into one 32-bit word.
- The NIC model function describing NIC register writes, *write_nic_register*, updates the NIC model state from *nic* to *nic'* as a result of writing *val* to the NIC register located at physical address *pa* in the NIC model state *nic*.

Under these four assumptions, there exists a transition from (*cpu*, *memory*, *nic*) to (*cpu'*, *memory*, *nic'*) with the label *CPU*.

Note that for a given state, only one of these three transition rules can be applied. The reason is that for the same arguments, *nic_access* is false if and only if both *cpu_read_nic_register* and *cpu_write_nic_register* return ⊥. In addition, if *cpu_read_nic_register* does not return ⊥, *cpu_write_nic_register* returns ⊥, and vice versa.

5.2.4.2.2 Transition Rules For NIC Transitions

Considering the implementation of the device model framework, there are three transition rules related to the autonomous transitions of the NIC model:

autonomous transitions without memory requests, autonomous transitions with memory read requests, and autonomous transitions with memory write requests.

The transition rule for autonomous transitions not issuing memory requests:

$$\frac{(nic', \bot, i) = nic\_execute(nic) \land cpu' = cpu[int \mapsto i]}{(cpu, memory, nic) \rightarrow_{NIC} (cpu', memory, nic')}.$$

The meaning of this transition rule is as follows. Assume the following:

- The NIC model performs an autonomous transition from *nic* to *nic'*, which does not issue a memory request (denoted by ⊥ in the second returned component).
- The CPU model state *cpu'* is equal to *cpu*, but where the interrupt flag, *cpu'.int*, reflects the interrupt status of the NIC model in the state *nic'* (denoted by *i* in the third returned component of *nic_execute*).

Under these two assumptions, there exists a transition from (*cpu*, *memory*, *nic*) to (*cpu'*, *memory*, *nic'*) with the label *NIC*. This transition rule corresponds to step 1 of the function *advance_single* of the device model framework (see Subsection 2.4.1).

The transition rule for autonomous transitions issuing memory read requests:

$$\frac{\begin{array}{c}(nic', (pa, \bot), i) = nic\_execute(nic) \land val = memory(pa) \land \\ nic'' = memory\_byte(nic', (pa, val)) \land cpu' = cpu[int \mapsto i]\end{array}}{(cpu, memory, nic) \rightarrow_{NIC} (cpu', memory, nic'')}.$$

The meaning of this transition rule is as follows. Assume the following:

- The NIC model performs an autonomous transition from *nic* to *nic'*. This transition issues a memory read request to physical address *pa* (denoted by (*pa*, ⊥) in the second returned component of *nic_execute*). The transition does not updates the interrupt status of the NIC model, but the interrupt status of *nic'* is indicated by *i*.
- At the physical address *pa*, *memory* contains the byte value *val*.
- When the NIC model in the state *nic'* is given a memory read request reply with the byte value *val*, read from the memory location with physical address *pa*, the NIC model transitions to the state *nic''*.
- The CPU model state *cpu'* is equal to *cpu*, possibly except for the interrupt flag which in *cpu'* is equal to the interrupt status of the NIC model in the state *nic'* (and *nic''* since memory read request reply transitions do not cause the NIC model to change the interrupt status).

Under these four assumptions, there exists a transition from (*cpu*, *memory*, *nic*) to (*cpu'*, *memory*, *nic''*) with the label *NIC*.

The transition rule for autonomous transitions issuing memory write requests:

$$\frac{\begin{array}{c}(nic', (pa, val), i) = nic\_execute(nic) \land val \neq \bot \\ memory' = memory[pa \mapsto val] \land cpu' = cpu[int \mapsto i]\end{array}}{(cpu, memory, nic) \rightarrow_{NIC} (cpu', memory', nic')}.$$

The meaning of this transition rule is as follows. Assume the following:

- The NIC model performs an autonomous transition from *nic* to *nic'*. This transition issues a memory write request to write the byte value *val* to the memory location at physical address *pa* (denoted by (*pa*, *val*) and $val \neq \bot$ in the second returned component of *nic_execute* and in the second conjunct of the premise, respectively). Similarly to transitions issuing memory read requests, this transition also does not update the interrupt status of the NIC model, but the interrupt status of the NIC model in the state *nic'* is indicated by *i*.
- *memory'* is equal to *memory*, possibly except for the argument *pa*, which *memory'* maps to the byte value *val*.
- *cpu'* is equal to *cpu*, but with the interrupt flag being equal to the interrupt status of the NIC model in the state *nic'*.

Under these three assumptions, there exists a transition from (*cpu*, *memory*, *nic*) to (*cpu'*, *memory*, *nic'*) with the label *NIC*.

These last two transition rules, which describe how the real model handles memory requests issued by the NIC model, correspond to steps 1 and 2 of *advance_single* (see Subsection 2.4.1).

5.2.4.3 Labeled Transition System of the Real Model

This subsection defines a four tuple that the reasoning in the proof plan uses to refer to the real model. That four tuple is defined in terms of the labeled transition system LTS_{real}:

$$LTS_{real} \overset{\text{def}}{=} (S_{real}, IS_{real}, L_{real}, \delta_{real}).$$

Each component of LTS_{real} is defined as follows:

- S_{real}: The set of states in LTS_{real}. That is, all possible instantiations of the data type real_state.
- $IS_{real} \subseteq S_{real}$: The set of initial states in LTS_{real}. Let $r \in S_{real}$. Intuitively, *r* is in IS_{real} if and only if the next instruction execution of the CPU model from the state *r* is the first execution of an instruction located in the memory region allocated to Linux. The states in IS_{real} describe the hardware states that hardware system executions enter after they have executed the initialization code of the hypervisor and are just about to start the execution of Linux. Hardware system executions start from hardware states that the hardware enter when it is powered on.

  Formally, *r* is in IS_{real} if and only if the following four conditions hold:

  1. There exists an execution trace $\pi = r_0 \rightarrow_{l\_0} r_1 \rightarrow_{l\_1} \ldots \rightarrow_{l\_n-1} r_n$, $n \geq 0$, generated by the device model framework instantiated with the NIC

model, such that r_0 describes a hardware state that the hardware might enter when it is powered on.

2. There exists a state r_k in π that is equal to r: $\exists 0 \leq k \leq n.\ r_k = r$.
3. In the state r, the MMU model maps the value of the program counter as executable to a physical address that is allocated to Linux: $LCE(r)$.

   LCE (Linux Code is Executed) is formally defined in Section 6.1 by means of the MMU model, and is used in the proof plan to denote when the CPU model executes the binary code of Linux:

   $$\text{bool } LCE(\text{real\_state } r).$$

4. For all states r_l preceding r_k in π, the MMU model does not map the value of the program counter as executable to a physical address allocated to Linux: $\forall 0 \leq l < k.\ \neg LCE(r_l)$.

- $L_{real} \overset{\text{def}}{=} \{CPU, EXC, RET, NIC\}$: The set of labels for the transitions in LTS_{real}. All transitions generated by the transition rules described in Subsection 5.2.4.2 have a label in L_{real}.
- $\delta_{real} \subseteq S_{real} \times L_{real} \times S_{real}$: The set of transitions in LTS_{real}. A transition (r, l, r') is written as $r \rightarrow_l r'$. $r \rightarrow_l r'$ is in δ_{real} if and only if $r \in S_{real}$ and $r \rightarrow_l r'$ can be generated by a transition rule described in Subsection 5.2.4.2.

The real model is a transition system that is formally defined in HOL4 by the device model framework instantiated with a HOL4 implementation of the NIC model described in Subsection 5.1 and Appendix C, and with the binary images of the hypervisor, the monitor and Linux. In this thesis, the real model is referred to by means of the four tuple $RM \overset{\text{def}}{=} (S, IS, \delta, \Pi)$. The components of *RM* are defined by means of LTS_{real} as follows (Figure 37 gives a graphical illustration of the real model):

- $RM.S \subseteq S_{real}$: The set of states in the real model. A real state r is in $RM.S$ if and only if there exists an execution trace $r_0 \rightarrow_{l\_0} r_1 \rightarrow_{l\_1} \ldots \rightarrow_{l\_n\text{-}1} r_n$, $n \geq 0$, such that:
  - The execution trace starts in a state from which Linux starts its execution: $r_0 \in IS_{real}$.
  - All transitions in the trace are generated by the transition rules described in Subsection 5.2.4.2: $\forall 0 \leq j < n.\ r_j \rightarrow_{l\_j} r_{j+1} \in \delta_{real}$.
  - There exists a state in the trace that is equal to r: $\exists 0 \leq j \leq n.\ r_j = r$.

  Hence, each state in $RM.S$ is reachable from an initial state from which Linux starts its execution.
- $RM.IS \overset{\text{def}}{=} IS_{real}$: The set of initial states of the real model. The execution of Linux starts from a state in $RM.IS$. Note that $RM.IS \subseteq RM.S$.
- $RM.\delta \subseteq S_{real} \times L_{real} \times S_{real}$: The set of transitions in the real model. $r \rightarrow_l r'$ is in $RM.\delta$ if and only if $r \in RM.S$ and a transition rule described in Subsection 5.2.4.2 can generate $r \rightarrow_l r'$, $r \rightarrow_l r' \in \delta_{real}$. Note that $r' \in RM.S$.
- $RM.\Pi$: The set of execution traces in the real model. Let

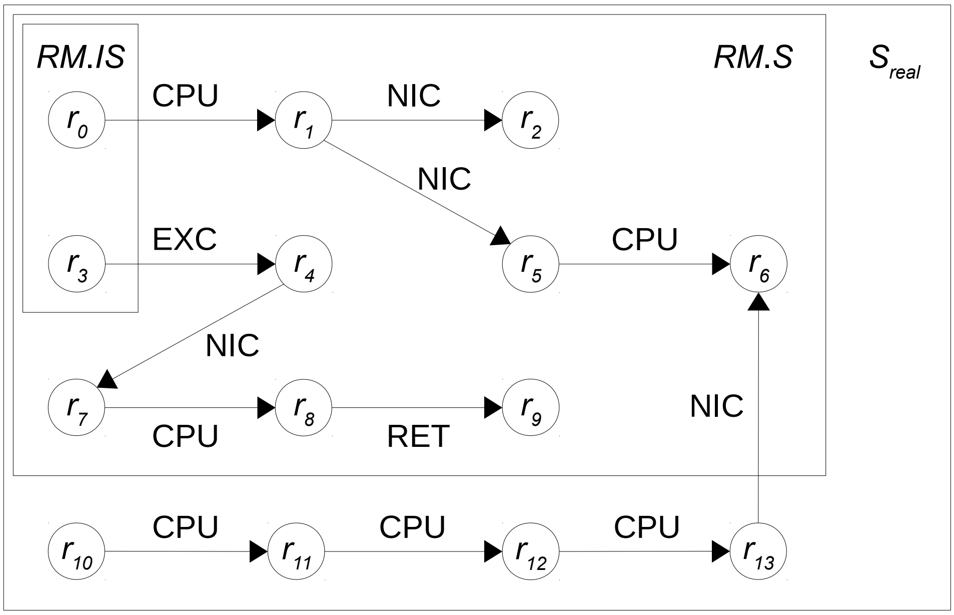


Figure 37: A graphical illustration of the roles of the four components of RM. The four tuple RM is used to denote the real model. RM.IS denotes the set of initial states in the real model. In this figure, RM.IS contains r_0 and r_3. RM.S denotes the set of states in the real model. In this figure, RM.S contains all states except r_{10}, r_{11}, r_{12} and r_{13}. The states r_{10}, r_{11}, r_{12} and r_{13} are all instantiations of the data type real_state but since they are not reachable from an initial state in the real model, they are not included in the real model. Hence, the real model only includes states that are reachable from initial states. RM.δ denotes the set of transitions in the real model. Each such transition is between two states that are included in the real model. In this figure, RM.δ contains all transitions except $r_{10} \rightarrow_{CPU} r_{11}$, $r_{11} \rightarrow_{CPU} r_{12}$, $r_{12} \rightarrow_{CPU} r_{13}$ and $r_{13} \rightarrow_{NIC} r_6$. RM.Π denotes the set of execution traces in the real model. Each such execution trace consists of a sequence of transitions starting from an initial state and in which each transition is in the real model. In this figure, RM.Π contains all execution traces starting from r_0 or r_3. An example of such an execution trace is $r_0 \rightarrow_{CPU} r_1 \rightarrow_{NIC} r_5$. RM.S, RM.δ and RM.Π are defined by means of the transition rules described in Subsection 5.2.4.2.

$$\pi = r_0 \rightarrow_{l\_0} r_1 \rightarrow_{l\_1} \ldots \rightarrow_{l\_n-1} r_n,\ n \geq 0.$$

$\pi \in RM.\Pi$ if and only if $r_0 \in RM.IS$ and each transition in π has been generated by some transition rule described in Subsection 5.2.4.2:

$$\forall 0 \leq j < n.\ r_j \rightarrow_{l\_j} r_{j+1} \in \delta_{real}.$$

Note that all transitions in traces in *RM.Π* are in *RM.δ*, and that all traces in *RM.Π* start from an initial state in *RM.IS*.

Figure 37 and its caption provides an intuition of the roles of the four components of *RM*. It is upon these four components of *RM* that the proof plan describes how it can be formally proved that only signed Linux code is executed. Appendix D provides an example of how an execution trace in *RM* can be generated by means of the transition rules that are described in Subsection 5.2.4.2, and how that trace operates on the visited states.

The MMU model [86], being a part of the real model, is denoted by *mmu*:

word32 $\cup$ $\{\perp\}$ *mmu*(real_state *r*, {*PL0*, *PL1*} *pl*, word32 *va*, {*rd*, *wt*, *ex*} *ar*).

The arguments of *mmu* are a real state *r*, a privilege level *pl* that is either equal to *PL0* or *PL1*, a virtual address *va* to translate to a physical address, and the type of the requested access *ar*, signifying either a read, write or execute access request. *mmu* performs a translation table walk on the real state *r* as the physical MMU performs a translation table walk on the hardware state described by *r*. *mmu* can be thought of as operating in three steps. First, *mmu* checks whether there exists a page table entry that maps the virtual address *va*. Second, *mmu* checks whether the access permissions specified by the page table entry (if found in the first step) and the value of the relevant field of the DACR register (stored in *r.cpu.cp15.DACR*) are compatible with the given privilege level *pl* and access request *ar*. Third, if the checks in the first two steps succeeded, the physical address mapped by the identified page table entry is returned. Otherwise $\perp$ is returned.

5.3 Ideal Model

As mentioned in the opening of this chapter, the proof plan in Chapter 6 is based on the simulation proof method. The simulation proof method can be used to prove that the behavior of one less abstract model is similar to the behavior of another more abstract model. If it is proved that the more abstract model satisfies a desired property and the less abstract model behaves similarly to the more abstract model, it can be proved that the desired property holds on the less abstract model, since the two models behave similarly. If the less abstract model describes the behavior of an implementation, it can be concluded that the implementation has the desired property. In the context of the proof plan, the less abstract model is the real model, the more abstract model is the ideal model, and the desired property is only signed Linux code is executed.

To prove that only signed Linux code is executed on the real model by means of the simulation proof method and the ideal model, the ideal model is defined as a transition system similar to the real model, but with one main difference that is related to the binary interface. (In this context of Linux, the binary interface is the interface between the binary code of Linux and the operations performed by: non-privileged CPU execution, the execution of the exception handlers, and the NIC. That is, the binary interface is the interface between the binary code of Linux and the operations that affect the execution of the binary code of Linux.) The ideal model describes the following binary interface:

- Non-privileged CPU execution operates as specified by the ARMv7 ISA. That is, in the ideal model, the CPU model in non-privileged mode executes as specified by the ARMv7 ISA.
- Exception handler execution operates as specified by the software design of the hypervisor and the monitor, as opposed to their implementation as is the case in the real model. That is, in the ideal model, privileged CPU

execution operates as specified by the software design of the hypervisor and the monitor. This formal description provided by the ideal model of how the exception handlers in the ideal model operates is the formal software design of the hypervisor and the monitor.

- The NIC operates as specified by the NIC specification.

To prove that only signed Linux code is executed on the real model by means of the simulation proof method and the ideal model, the following steps can be taken:

1. Prove that only signed Linux code is executed on the binary interface described by the ideal model: That is, the formal software design of the hypervisor and the monitor provided by the ideal model is proved to be correct. Once it has been proved that the formal software design is correct, the formal software design can be considered to be the formal specification of the implementation of the hypervisor and the monitor. (Considering the explanation in the first paragraph of this section, this step corresponds to proving the desired property on the more abstract model.)
2. Prove that the execution of the binary code of the hypervisor and the monitor operates according to their formal specification provided by the ideal model: That is, the implementation of the hypervisor and the monitor is proved to be correct with respect to their formal specification.
3. Prove that the executions of the binary code of Linux on the binary interfaces described by the ideal model and the real model are identical: To prove this property, it must be proved that the binary interfaces described by the ideal model and the real model are identical. Since the ideal model and the real model describe the binary interface of the exception handlers differently (formal software design and implementation, respectively), it must be proved that the binary interfaces of the exception handlers described by the ideal model and the real model are identical. The proof in this third step can therefore utilize the proof in step 2. (This step corresponds to proving that the less abstract model behaves similarly to the more abstract model.)

Step 3 enables the property proved on the ideal model in step 1 to be transferred from the ideal model to the real model. Hence, only signed Linux code is executed on the binary interface described by the real model. The conclusion is therefore that only signed Linux code is executed in a system consisting of an ARMv7 CPU, the NIC on Beaglebone Black, the hypervisor and the monitor. The proof plan describes in detail how these proofs can be constructed and how they fit together. The following describes the ideal model and decisions behind its design from the perspective of the three proof steps listed above.

To ease the construction of the proof of that only signed Linux code is executed on the binary interface described by the ideal model (proof step 1), the ideal model describes the exception handlers of the binary interface to be executed *atomically* in one transition, as opposed to as in the real model to be executed non-atomically in several transitions. These atomic exception handler transitions in the ideal model occur only when the CPU model is in privileged mode and the non-deterministic scheduler of the device model framework schedules the CPU model to perform the next transition of the ideal model. The operations described by these atomic exception handler transitions modify the state as described by the software design of the hypervisor and the monitor. First, the exception is handled (e.g. assigning CPU registers with certain values for forwarding an interrupt to Linux or handling a memory mapping request). Then, the CPU model is restored to non-privileged mode, from where the execution of Linux can continue. Figure 38 illustrates the difference between how the ideal model and the real model describe executions of the exception handlers.

The operations performed by each exception handler in the ideal model are specified by a mathematical function (as will be seen in subsection 5.3.2.2). To further simplify the construction of the proof in step 1, in the ideal model these functions read and write the values of the data structures of the software design by reading and writing *abstract state components*, as opposed to reading and writing the CPU register and memory state components (which describe the content of the physical CPU registers and memory). (This way of storing data structure values in the ideal model is described in Subsection 5.3.1.)

To enable the construction of the proof of that the execution of the binary code of the hypervisor and the monitor operates according to their formal software design (proof step 2), the ideal model specifies the formal software design as being the binary interface of the exception handlers. There are two reasons why the ideal model specifies the formal software design in this way.

First, the ideal model specifies the formal software design of the hypervisor and the monitor to be a *set of exception handlers*. The reason is that the hypervisor is only invoked when exceptions occur and it invokes the monitor, allowing the monitor to also be considered as being a part of the exception handlers. This one-to-one correspondence of when the operations of the formal software design of the hypervisor and the monitor are performed in the ideal model, and when the operations of the implementations of the exception handlers of the hypervisor and the monitor are performed in the real model, is what makes the ideal model usable in the application of the simulation proof method. In other words, considering the formal software design of the hypervisor and the monitor provided by the ideal model as being their formal specification (as motivated in the first step in the proof list above), this way of specifying their formal specification in the ideal model is what makes it possible to prove by means of the simulation proof method and the ideal model that their implementation is correct with respect to their formal specification. Figure 38 and the second half of its caption elaborates this reasoning.

Second, the ideal model specifies the formal software design to be *all exception handlers*. In an execution trace, irrespectively of whether the execution trace is in the ideal model or in the real model, any exception can potentially occur. If an exception handler is unspecified by the ideal model, the handling of that exception could be considered to perform any operations, potentially producing a state from which unsigned Linux code can be executed. In such a case, it would not be possible to prove that only signed Linux code is executed on the binary interface described by the ideal model (proof step 1). Also, exception handlers that are left unspecified by the ideal model could be considered to be allowed to be implemented arbitrarily by the hypervisor and the monitor. Implementations of

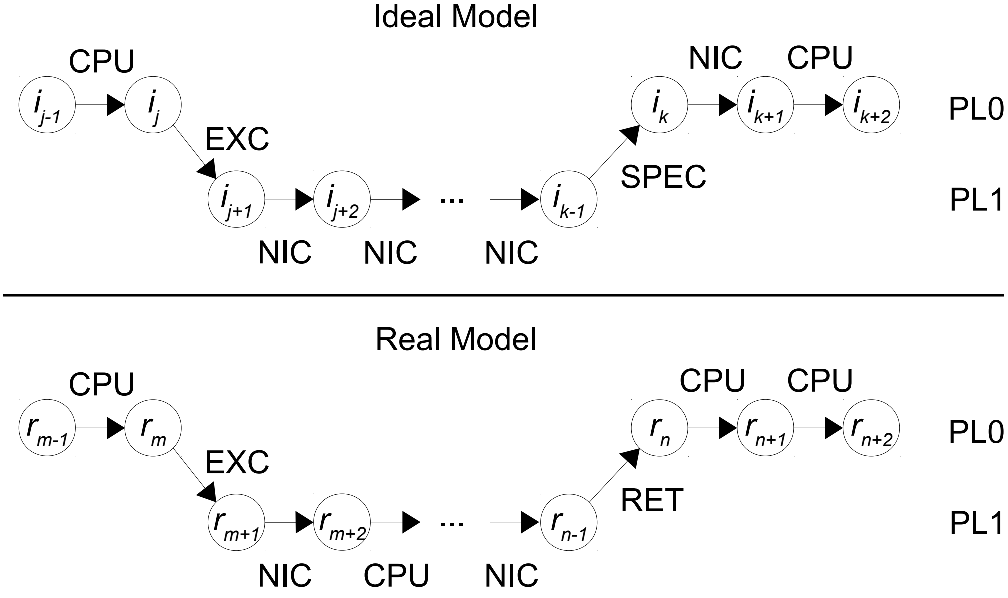


Figure 38: An illustration of the difference between how the ideal model and the real model describe executions of the exception handlers. In the ideal model, the execution of an exception handler is described by one single specification transition. NIC transitions can therefore not occur during the execution of an exception handler in the ideal model. The specification transitions describing the execution of the exception handlers in the ideal model specify how the exception handlers implemented by the hypervisor and the monitor shall operate. Hence the name specification transition and their label being SPEC. Since some exception handlers read or write NIC registers, SPEC transitions may include NIC register read or write transitions of the NIC model. In the real model, the execution of an exception handler is described by several CPU transitions, possibly intermingled with NIC transitions. Since the exception handler execution shown in the lower part of the figure in the real model only visits states in which the CPU model is in privileged mode and the monitor is executed in non-privileged mode, that exception handler execution does not include monitor execution. Proving, by means of the simulation proof method and the ideal model, that an implementation of an exception handler of the hypervisor and the monitor operates according to its formal specification, is done by proving that the execution traces in the real model of that implementation operate as the execution traces in the ideal model of the formal specification of that exception handler. Considering the figure, it must be proved that the operations performed by the execution trace in the real model between the real states r_{m+1} and r_n, are identical to the operations performed by the execution trace in the ideal model between the ideal states i_{j+1} and i_k. This involves proving that the CPU and RET transitions between r_{m+1} and r_n together operate as the SPEC transition $i_{k-1} \rightarrow_{SPEC} i_k$, and that the NIC transitions between r_{m+1} and r_{n-1} operate as the NIC transitions between i_{j+1} and i_{k-1}. For such a proof to succeed, the implementation of the exception handler must operate as formally specified by the ideal model, and each NIC model automaton must perform the same operations in the two sets of transitions. Which operations each NIC model automaton performs depend on the scheduling of the device model framework and the NIC model and on the non-determinism of the NIC model automata.

unspecified exception handlers could then enable unsigned Linux code to be executed on the binary interface described by the real model and therefore on the hardware. Hence, it would be impossible to prove that only signed Linux code is executed.

All exception handlers must therefore be specified by the ideal model. The software design described in Chapter 3 specifies only the memory mapping and NIC register write request handlers, which are only a part of the supervisor call and data abort exception handlers, respectively. The formal software design that the ideal model specifies is therefore not the software design described in Chapter 3, but instead an extension of it specifying all exception handlers. Even though each exception handler must not enable execution of unsigned Linux code, this thesis focuses only on the memory mapping and NIC register write request handlers, since they are the most critical parts of the exception handlers for ensuring that only signed Linux code is executed.

To enable the construction of the proof of that the executions of the binary code of Linux are identical on the binary interfaces described by the ideal model and the real model (proof step 3), in the ideal model the CPU model in non-privileged mode executes according to the ARMv7 ISA and the NIC model operates according to the NIC specification. The binary interfaces of (i) the non-privileged CPU execution and (ii) the operation of the NIC are therefore identical in the ideal model and the real model, respectively. These are the two similarities between the ideal model and the real model. Due to these two similarities, states in the ideal model include all state components that are included by states in the real model. Hence, the data type of the states in the ideal model (denoted ideal_state) is an extension of the data type of the states in the real model (denoted real_state). The extension is the state components holding the values of the data structures of the formal software design of the hypervisor and the monitor.

To summarize, the ideal model is a transition system that is identical to the transition system of the real model, except in two respects. First, in the ideal model the exception handlers are executed atomically in one transition, as opposed to non-atomically in several transitions as is the case in the real model. Second, in the ideal model the values of all data structures of the exception handlers are stored in dedicated state components, as opposed to in the CPU register and memory state components as is the case in the real model. It is because of these two differences that the ideal model is more abstract than the real model.

Since the proof plan shall be implemented in HOL4, and it is based on the ideal model, the ideal model must be implemented in HOL4. Significant parts of that implementation can be guided by:

- The HOL4 implementation of the original formal software design of the memory mapping request handlers [86], and the formal descriptions in Section B.2 of the NIC related extensions of these handlers.
- The pseudocode of the NIC register write request handlers.
- The HOL4 implementations of the device model framework [85], the ARMv7 ISA model [84] and the ARMv7 MMU model [86].

- The pseudocode of the NIC model.

The following subsections formally describe the ideal model similarly to how Subsection 5.2.4 formally describes the real model. Subsection 5.3.1 briefly describes the data type of the states in the ideal model. Subsection 5.3.2 describes the transition rules of the ideal model. Subsection 5.3.3 defines the four tuple denoting the ideal model.

5.3.1 States in the Ideal Model

A state in the ideal model has the data type ideal_state, an instance of which is called an ideal state. The set of all ideal states is denoted by S_{ideal}. An ideal state *i* in S_{ideal} has the shape *i* = (*cpu*, *memory*, *nic*, *spec*). The state components *cpu*, *memory* and *nic* have the same roles as in a real state. The state component *spec* is the abstract state component mentioned above. It has the data type spec_state and contains the values of all data structures used in the formal software design specified by the ideal model. Since the focus of this thesis is on the memory mapping and NIC register write request handlers, this thesis only refers to the data structures related to those handlers. (Those data structures are described in Section 3.4.) For instance, *i.spec.tx0_active_queue* denotes the value of the variable *tx0_active_queue* in the state *i*. The relevant parts of ideal_state and spec_state are formally defined in Section B.1.

5.3.2 Transition Rules of the Ideal Model

The ideal model has three types of transitions:

- Non-privileged CPU transitions: Each of these transitions describes the execution of one CPU instruction in non-privileged mode. Since the execution of the operations of the formal specification (software design) of the hypervisor and the monitor are described by the specification transitions, these transitions describe executions of CPU instructions located in the physical memory region allocated to Linux.
- Specification transitions: Each of these transitions describes the atomic execution of the operations of the formal specification of the hypervisor and the monitor.
- NIC transitions: Is one autonomous transition of the NIC model, and possibly one following memory read request reply transition if the autonomous transition issued a memory read request.

The transition rules that describe these three types of transitions are described in the following three subsections.

5.3.2.1 Transition Rules for Non-Privileged CPU Transitions

The transition rules describing non-privileged CPU transitions in the ideal model are nearly identical to the transition rules describing CPU transitions in the real model. The difference is that the non-privileged CPU transition rules of the ideal model have a conjunct in the premise requiring the CPU model to be in non-privileged mode in the state from which a described transition starts. This requirement is needed because non-privileged CPU transitions start only from states in which the CPU model is in non-privileged mode. In addition, since only the specification transitions operate on the *spec* state component, the *spec* state component is ignored (unmodified) by the non-privileged CPU transition rules.

The transition rules for non-privileged CPU transitions follow:

- No NIC register access:

$$\frac{\begin{array}{c} mode(cpu) = usr \wedge (cpu', memory') = cpu\_execute(cpu, memory) \wedge \\ \neg nic\_access(cpu, memory) \end{array}}{(cpu, memory, nic, spec) \rightarrow_{cpu\_exec\_type(cpu,\, cpu')} (cpu', memory', nic, spec)}.$$

- NIC register reads:

$$\frac{\begin{array}{c} pa = cpu\_read\_nic\_register(cpu, memory) \wedge \\ (nic', val) = read\_nic\_register(nic, pa) \wedge \\ memory' = memory[pa \mapsto val[7{:}0], \\ pa + 1 \mapsto val[15{:}8], pa + 2 \mapsto val[23{:}16], pa + 3 \mapsto val[31{:}24]] \wedge \\ mode(cpu) = usr \wedge (cpu', memory'') = cpu\_execute(cpu, memory') \end{array}}{(cpu, memory, nic, spec) \rightarrow_{CPU} (cpu', memory, nic', spec)}.$$

- NIC register writes:

$$\frac{\begin{array}{c} pa = cpu\_write\_nic\_register(cpu, memory) \wedge \\ mode(cpu) = usr \wedge (cpu', memory') = cpu\_execute(cpu, memory) \wedge \\ val = memory'(pa + 3) :: memory'(pa + 2) :: \\ memory'(pa + 1) :: memory'(pa) \wedge \\ nic' = write\_nic\_register(nic, pa, val) \end{array}}{(cpu, memory, nic, spec) \rightarrow_{CPU} (cpu', memory, nic', spec)}.$$

5.3.2.2 Transition Rules for Specification Transitions

Let the following functions denote the formal specification of all exception handlers of the hypervisor and the monitor: *fiq_interrupt*, *irq_interrupt*, *supervisor_call*, *undefined_instruction*, *prefetch_abort* and *data_abort*. (Subsection 2.3.1 describes all exceptions of an ARMv7 CPU, and Subsection 2.3.4.1 briefly describes how the exceptions are handled by the hypervisor.) The argument and return values of these functions have the data type ideal_state.

In Sections 3.5 and 3.6, the functions denoting the memory mapping and NIC register write request handlers are shown to operate on ideal states. The reason why those handler functions are shown to operate on ideal states is because they are applied by *supervisor_call* and *data_abort*, which operate on ideal states. Also, a formal definition of *data_abort* is partially given in Section B.3. That formal definition of *data_abort* describes how the NIC register write request handlers are invoked and operate when the data abort exception handler is invoked because Linux attempted to write a NIC register. Furthermore, Section B.5 discusses why the transitions specified by the formal software design are atomic.

The transition rules that describe the specification transitions are:

- Exceptions causing the CPU model to enter fiq, irq, svc or und mode:

$$\frac{\begin{array}{c} \mathit{mode}(\mathit{cpu}) = e \wedge \\ (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}') = H_{(e)}(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \end{array}}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{SPEC} (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}')},$$

  where $e \in \{\mathit{fiq}, \mathit{irq}, \mathit{svc}, \mathit{und}\}$, and $H_{(e)}$ is the function that denotes the formal specification of the exception handler that handles exception e. For instance, $H_{(\mathit{fiq})} = \mathit{fiq\_interrupt}$.

- Prefetch abort exceptions:

$$\frac{\begin{array}{c} \mathit{mode}(\mathit{cpu}) = \mathit{abt} \wedge \mathit{cpu.uregs.r15} = \mathrm{0xFFFF000C} \wedge \\ (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}') = \mathit{prefetch\_abort}(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \end{array}}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{SPEC} (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}')}.$$

- Data abort exceptions:

$$\frac{\begin{array}{c} \mathit{mode}(\mathit{cpu}) = \mathit{abt} \wedge \mathit{cpu.uregs.r15} = \mathrm{0xFFFF0010} \wedge \\ (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}') = \mathit{data\_abort}(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \end{array}}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{SPEC} (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec}')}.$$

5.3.2.3 Transition Rules for NIC Transitions

The transition rules for NIC transitions are identical to their real model correspondence with the difference that the states in the conclusion of the rules are extended with the *spec* state component, which is always unmodified:

- The NIC does not issue a memory access request:

$$\frac{(\mathit{nic}', \bot, i) = \mathit{nic\_execute}(\mathit{nic}) \wedge \mathit{cpu}' = \mathit{cpu}[\mathit{int} \mapsto i]}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{NIC} (\mathit{cpu}', \mathit{memory}, \mathit{nic}', \mathit{spec})}.$$

- The NIC issues a memory read request:

$$\frac{\begin{array}{c} (\mathit{nic}', (\mathit{pa}, \bot), i) = \mathit{nic\_execute}(\mathit{nic}) \wedge \mathit{val} = \mathit{memory}(\mathit{pa}) \wedge \\ \mathit{nic}'' = \mathit{memory\_byte}(\mathit{nic}', (\mathit{pa}, \mathit{val})) \wedge \mathit{cpu}' = \mathit{cpu}[\mathit{int} \mapsto i] \end{array}}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{NIC} (\mathit{cpu}', \mathit{memory}, \mathit{nic}'', \mathit{spec})}.$$

- The NIC issues a memory write request:

$$\frac{\begin{array}{c} (\mathit{nic}', (\mathit{pa}, \mathit{val}), i) = \mathit{nic\_execute}(\mathit{nic}) \wedge \mathit{val} \neq \bot \\ \mathit{memory}' = \mathit{memory}[\mathit{pa} \mapsto \mathit{val}] \wedge \mathit{cpu}' = \mathit{cpu}[\mathit{int} \mapsto i] \end{array}}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}, \mathit{spec}) \rightarrow_{NIC} (\mathit{cpu}', \mathit{memory}', \mathit{nic}', \mathit{spec})}.$$

5.3.3 Labeled Transition System of the Ideal Model

The ideal model is denoted similarly to how the real model is denoted. Consider the labeled transition system

$$LTS_{ideal} = (S_{ideal}, IS_{ideal}, L_{ideal}, \delta_{ideal}),$$

where each component is:

- S_{ideal}: The set of all possible instantiations of the data type ideal_state.
- $IS_{ideal} \overset{\text{def}}{=} \{i \mid SEC(i) \land NIC\_INIT(i.nic) \land SPEC\_INIT(i.spec) \land LCE(i)\}$:

  The set of initial states in LTS_{ideal}. Intuitively, an ideal state is in IS_{ideal} if and only if: (i) only signed Linux code can be executed from that state, (ii) the NIC model and the data structures of the formal software design are correctly initialized in that state, and (iii) Linux starts its execution from that state. Hence, it is from the initial states of LTS_{ideal} that the execution of Linux starts, and in which all state components are in a state such that only signed Linux code can be executed.

  The predicates used in the definition of IS_{ideal} have the following meaning (this list can also be used as a reference when reading the proof plan):

  - bool *SEC*(ideal_state *i*): A security invariant implying that if an ideal state *i* satisfies it, then (i) no transition from *i* can falsify *SEC*, and (ii) if the CPU executes a Linux instruction from *i* then that instruction is located in a memory block whose signature is in the golden image. *SEC* is partly described in the proof plan and is formally defined in completion in Appendix E.
  - bool *NIC_INIT*(nic_state *nic*): Is true if and only if *nic* reflects a physical NIC that has just been powered on and then initialized (the DMA hardware has been reset and the four HDP and CP registers have been zeroed). Hence, *nic* reflects a physical NIC that is in a secure and idle state. The comments in the formal definition of nic_state in Section C.3 describe which values *nic* shall have in order to satisfy *NIC_INIT*.
  - bool *SPEC_INIT*(spec_state *spec*): Is true if and only if the values of the state components of *spec* reflect correctly initialized data structures of the formal software design described in Section 3.4. The initial values of those data structures are described by the comments in the formal definition of spec_state in Section B.1. Those data structure values make the formal software design operate as if in a corresponding hardware state, the physical NIC is in an initialized state and Linux is just to start its execution.
  - bool *LCE*(ideal_state *i*): Is true if and only if in the ideal state *i*, *mmu* maps the value of the program counter as executable to a physical address allocated to Linux. (*mmu* and *LCE* are defined for ideal states as they are defined for real states as described in Subsection 5.2.4.3. *LCE* is formally defined for real states in the Section 6.1.)

  SEC, *NIC_INIT* and *SPEC_INIT* must be consistent with each other since they depend on common variables.

- $L_{ideal} \overset{\text{def}}{=} \{CPU, EXC, SPEC, NIC\}$: The transitions in δ_{ideal} have a label in L_{ideal}. Compared to L_{real}, *RET* is replaced by *SPEC* since only specification transitions describe exception returns in the ideal model.

- $\delta_{ideal} \subseteq S_{ideal} \times L_{ideal} \times S_{ideal}$: A transition $i \rightarrow_l i'$ is in δ_{ideal} if and only if $i \rightarrow_l i'$ can be generated by a transition rule described in Subsection 5.3.2.

The ideal model is a transition system that shall be defined in HOL4. That definition consists of the device model framework instantiated with a HOL4 implementation of the NIC model, and a modification the CPU model such that it describes the operations of the formal specification of all exception handlers. The proof plan refers to the ideal model by means of the four tuple $IM \overset{\text{def}}{=} (S, IS, \delta, \Pi)$. The four components of IM are defined on top of LTS_{ideal} as follows:

- $IM.S \subseteq S_{ideal}$: The set of states in the ideal model. An ideal state i is in $IM.S$ if and only if there exists an execution trace $i_0 \rightarrow_{l\_0} i_1 \rightarrow_{l\_1} \ldots \rightarrow_{l\_n\text{-}1} i_n$, $n \geq 0$, such that:
    - The execution trace starts in a state in which the NIC model and the data structures of the formal software design are initialized and from which the execution of signed Linux code starts: $i_0 \in IS_{ideal}$.
    - All transitions in the trace are generated by the transition rules described in Subsection 5.3.2: $\forall 0 \leq j < n.\ i_j \rightarrow_{l\_j} i_{j+1} \in \delta_{ideal}$.
    - There exists a state in the trace that is equal to i: $\exists 0 \leq j \leq n.\ i_j = i$.
- $IM.IS \overset{\text{def}}{=} IS_{ideal}$: The set of initial states in the ideal model, from which the execution of signed Linux code starts.
- $IM.\delta \subseteq S_{ideal} \times L_{ideal} \times S_{ideal}$: The set of transitions in the ideal model. A transition $i \rightarrow_l i'$ is in $IM.\delta$ if and only if $i \in IM.S$ and there is a transition rule in Subsection 5.3.2 that can generate that transition, $i \rightarrow_l i' \in \delta_{ideal}$.
- $IM.\Pi$: The set of execution traces in the ideal model. Let

  $$v = i_0 \rightarrow_{l\_0} i_1 \rightarrow_{l\_1} \ldots \rightarrow_{l\_n\text{-}1} i_n,\ n \geq 0.$$

  $v \in IM.\Pi$ if and only if $i_0 \in IM.IS$ and each transition in v has been generated by some transition rule described in Subsection 5.3.2:

  $$\forall 0 \leq j < n.\ i_j \rightarrow_{l\_j} i_{j+1} \in \delta_{ideal}.$$

Also, the signature function *sign* is a part of the ideal model. *sign* specifies how the signature of the content of a memory block is computed:

$$\text{wordx } sign(\text{word32768 } code).$$

The argument *code* is a bit string of length 32768, which is equal to 4 kB. The return value is the signature of the bit string *code*, which is a bit string of length x. x is left unspecified and shall be replaced by the actual bit length of the signatures. For instance, if SHA-256 signatures are used, x is equal to 256. *sign* specifies the implementation of the signature function of the monitor. The monitor invokes the implemented signature function to check whether the signature of the content of a memory block allocated to Linux is in the golden image. If that is the case, the content of the block is considered signed and can be mapped as executable, assuming the block is not writable by the physical CPU when executing Linux code and not writable by the physical NIC.

The proof plan reasons about the real model and the ideal model to describe how it can be formally proved that only signed Linux code is executed in a system consisting of an ARMv7 CPU, a memory, the NIC on BeagleBone Black, the hypervisor and the monitor. Since these models and hardware and software components have been formalized and explained, it is time to present the proof plan.

6 Proof Plan

This chapter presents a pen-and-paper description, called the proof plan, of how it can be formally proved in HOL4 that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed, in a system consisting of an ARMv7 CPU, a memory, the NIC on BeagleBone Black, the hypervisor, the monitor, and the paravirtualized Linux 3.10 kernel. The proof plan is based on the simulation proof method, the real model (described in Section 5.2), the ideal model (described in Section 5.3), and the software design of the hypervisor and the monitor (partly described in Sections 3.4, 3.5 and 3.6).

Having a plan for how to implement a formal proof in a theorem prover is necessary before the formal proof is implemented. Otherwise, it is easy for the prover to get lost in all details, and that the formal proof gets unstructured and difficult to understand because the theorem prover gets a significant impact on the proving procedure. Such impacts can cause the prover to perform significant amounts of unnecessary work.

The proof plan is structured by means of nine top-level lemmas (Lemma I-IX). The proof plan describes how these lemmas can be applied to prove that only signed Linux code is executed, and how the top-level lemmas can be proved. Some of the top-level lemmas are based on sub-level lemmas, which are described similarly, but more informally, in Appendix F.

This chapter is structured as follows. Section 6.1 describes definitions of constants and functions used in the proof plan (see Section 2.1 for a description of notation). Section 6.2 formally defines the goal of that only signed Linux code is executed in terms of the four tuple *RM*, denoting the real model. In addition, Section 6.2 defines all top-level lemmas and gives a high-level view of how they can be applied to prove the goal.

The following three sections describe the details of the proof plan. Section 6.3 describes how it can be proved that only signed Linux code is executed on the binary interface described by the ideal model, *IM*. (The second paragraph in Section 5.3 explains what the binary interface is in this context.) This includes a description of how it can be proved that the formal software design of the hypervisor and the monitor is correct. (This formal software design is an extension of the software design described in Sections 3.4, 3.5 and 3.6, and which describes all exception handlers of the hypervisor and the monitor; see also Section 5.3 for an explanation of this complete formal software design). A correctness proof of the formal software design allows the formal software design to not only be considered as a description of the exception handlers of the hypervisor and the monitor, but also as a formal specification for the binary code implementation of the hypervisor and the monitor.

Section 6.4 describes how it can be proved that the executions of the binary code of Linux on the binary interfaces described by the ideal model and the real model are identical. This includes a description of how it can be proved that the executions of the binary code implementation of the hypervisor and the monitor operate according to the formal specification provided by the ideal model. (In the beginning of Section 5.3, there is a numbered list of three proof steps. Referring to that numbered list, Section 6.3 describes proof step 1, and Section 6.4 describes proof step 3, which includes proof step 2.)

Section 6.5 finishes the proof plan by describing the final three top-level lemmas (Lemma VII-IX) that are applied to prove the goal, and derived from the first six top-level lemmas described in Sections 6.3 (Lemma I-III) and 6.4 (Lemma IV-VI). That is, these final three top-level lemmas are used to transfer the property of only signed Linux code execution from the ideal model to the real model, thereby proving the goal.

Section 6.6 discusses the correctness of the proof plan and whether it is practically feasible to implement the proof plan in HOL4. A summary of the proof plan is also presented.

In contrast to Chapter 5, in this chapter, no particular distinction is made between the hardware and the corresponding models, unless a description explicitly refers to a model. Formulas formulated in terms of functions referring to models are intended to describe a property involving the hardware, and in such cases, the reader can think about the hardware instead of the models.

6.1 Definitions of Constants and Functions

The following list defines the constants and functions used in the proof plan (this list can also be used as a reference when reading this chapter):

- *LINUX_MEM*: The set of all bytewise memory addresses of the physical memory region allocated to Linux.

- $\text{word32 } \mathit{memory\_word}(\text{word32} \rightarrow \text{word8 } \mathit{memory}, \text{word32 } \mathit{pa}) \overset{\text{def}}{=}$
  $\quad \text{return } \mathit{memory}(\mathit{pa} + 3) :: \mathit{memory}(\mathit{pa} + 2) ::$
  $\qquad \mathit{memory}(\mathit{pa} + 1) :: \mathit{memory}(\mathit{pa})$

  memory_word returns a 32 bit word located at physical address *pa* in the memory state *memory* in little-endian order.

- $\text{word32768 } \mathit{content}(\text{word32} \rightarrow \text{word8 } \mathit{memory}, \text{word20 } \mathit{bl}) \overset{\text{def}}{=}$
  $\quad \text{return } \mathit{memory}(\mathit{bl} :: 0^{12} + 4095) :: \mathit{memory}(\mathit{bl} :: 0^{12} + 4094) :: \ldots$
  $\qquad \ldots :: \mathit{memory}(\mathit{bl} :: 0^{12})$

  content returns the 32768 bits (which is 4 kB and the size of one block) contained in the memory block with index *bl*, where the memory is in the state *memory*. '::' and 0^{12} denotes concatenation and 12 consecutive zeros, respectively.

- $\text{<word32768> } \mathit{linux\_code}(\text{real\_state } r) \overset{\text{def}}{=}$
  $\quad \{\mathit{code} \mid \exists \mathit{va}, \mathit{pa} \in \text{<word32>}.\ \mathit{mmu}(r, \mathit{PL0}, \mathit{va}, \mathit{ex}) = \mathit{pa} \land$
  $\qquad \mathit{pa} \in \mathit{LINUX\_MEM} \land$
  $\qquad \mathit{code} = \mathit{content}(r.\mathit{memory}, \mathit{pa}[31{:}12]) \land$
  $\qquad r.\mathit{cpu}.\mathit{cp15}.\mathit{DACR}[5{:}4] = 0\text{b}01\}$

  linux_code returns the set of the contents of all blocks allocated to Linux and that are mapped as executable in non-privileged mode, in the real state *r*. (*mmu* is explained in Subsection 5.2.4.3.)

- bool *LCE*(real_state *r*) $\overset{\text{def}}{=}$
  $\exists pa \in LINUX\_MEM.$
  $mmu(r, PL0, r.cpu.uregs.r15, ex) = pa \lor$
  $mmu(r, PL1, r.cpu.uregs.r15, ex) = pa$

  LCE (Linux Code is Executed) returns true if and only if the next CPU instruction execution will succeed (takes effect without raising an exception, except for supervisor call instructions) and the executed instruction is located in the memory region allocated to Linux. That is, the virtual address contained in the program counter is mapped as executable to a physical address allocated to Linux. Intuitively, *r* is a state from which Linux code can be executed.

- bool *CPUL*(real_state *r*) $\overset{\text{def}}{=}$ $mode(r) = usr \land r.cpu.cp15.DACR$[5:4] = 0b01

  CPUL (CPU configured to execute Linux) is true if and only if the CPU (including the MMU whose operation is affected by DACR) is configured to execute Linux. The CPU is configured to execute Linux if and only if the CPU is in non-privileged mode, and the DACR register field dedicated to the physical memory region allocated to Linux (bits five and four of DACR) is set such that, the executions of Linux access Linux memory according to how they have configured the page tables. Subsection 2.3.1 explains DACR. The last paragraph in Subsection 2.3.4.1 explains how DACR is used by the hypervisor. Subsection 5.2.4.2.1 explains *mode*.

  The following explains the meanings of *LCE* and *CPUL* when applied to a real state *r*:

  - *LCE*(*r*) = *false* and *CPUL*(*r*) = *false*: The MMU does not map the virtual address in the program counter as executable to a physical address allocated to Linux, and the CPU is not configured to execute Linux. This means that the configurations of the page tables and the MMU are consistent with the configuration of the CPU to not execute Linux.
  - *LCE*(*r*) = *false* and *CPUL*(*r*) = *true*: The MMU does not map the virtual address in the program counter as executable to a physical address allocated to Linux, and the CPU is configured to execute Linux. A state *r* can satisfy this condition when, for instance, the CPU has executed an instruction that causes the program counter to be set to a virtual address that is not mapped as executable to a physical address allocated to Linux. Still, the configurations of the page tables and the MMU are consistent with the configuration of the CPU to execute Linux.
  - *LCE*(*r*) = *true* and *CPUL*(*r*) = *false*: The MMU maps the virtual address in the program counter as executable to a physical address allocated to Linux, and the CPU is not configured to execute Linux. This means that the configurations of the page tables and the MMU are not consistent with the configuration of the CPU to not execute Linux. In other words, the execution of the hypervisor has resumed the execution of Linux incorrectly. If the CPU is in privileged mode, this means that Linux has control of the system, which must not occur.

- ○ $LCE(r) = true$ and $CPUL(r) = true$: The MMU maps the virtual address in the program counter as executable to a physical address allocated to Linux, and the CPU is configured to execute Linux. This means that the configurations of the page tables and the MMU are consistent with the configuration of the CPU to execute Linux.

  Hence, what is wanted is that $LCE(r) \Rightarrow CPUL(r)$ holds for all $r \in RM.S$. How this formula can be proved is described in Subsection 6.5.1.

- spec_state *hvm_to_spec*(word32 → word8 *memory*)

  hvm_to_spec takes a state of the memory as argument and returns a spec_state instance. That spec_state instance contains the values of the data structures of the formal software design, implemented in the hypervisor and monitor and stored in their memory regions. The following examples illustrate the meaning of *hvm_to_spec*. Let $s = hvm\_to\_spec(memory)$, then:

  - ○ *s.tx0_active_queue*: Is equal to the 32-bit value stored in the hypervisor memory region that contains the value of the variable *tx0_active_queue* of the formal software design. For instance, if *tx0_active_queue* is located at physical address 0x4 and *memory_word*(*memory*, 0x4) = 0x2, then *s.tx0_active_queue* = 0x2.
  - ○ $s.\rho_{NIC}$(0x3): Is equal to the 32-bit value stored in the hypervisor memory region that contains the value of the data structure ρ_{NIC} indexed by 0x3. For instance, if the entry 0x3 of ρ_{NIC} is stored at physical address 0x8, and *memory_word*(*memory*, 0x8) = 0xA, then $s.\rho_{NIC}$(0x3) = 0xA.
  - ○ *s.GI*: The golden image containing the set of bit strings in the monitor memory that is used by the monitor to decide whether the signature of a block is valid. For instance, if the golden image stored in the monitor memory contains two signatures, each of 32 bits, stored at 0x0 and 0x4,

    memory_word(*memory*, 0x0) = 0x76543210, and

    memory_word(*memory*, 0x4) = 0x01234567,

    then *s.GI* = {0x76543210, 0x01234567}.

    Since the golden image is a part of the state it could potentially change during execution. In this proof plan, it is assumed that the golden image shall be constant. It is therefore necessary to prove a lemma stating that the golden image of the monitor is constant. No such lemma is included in the proof plan. However, as is mentioned in Section 8.1, the current implementation of the hypervisor and the monitor allows updates of the golden image. Such an implementation requires a different lemma stating that all modifications of the golden image are due to secure updates of the golden image.

- bool *LINUX_CODE_SIGNED*(real_state *r*) $\overset{\text{def}}{=}$
  $\forall code \in$ <word32768>.
  $code \in linux\_code(r) \Rightarrow sign(code) \in hvm\_to\_spec(r.memory).GI$

If a memory block is mapped as executable in non-privileged mode and allocated to Linux, then the contents of that block are signed. (*sign* is explained in Subsection 5.3.3.)

- bool *EXEC_SIGNED_LINUX_CODE*(real_state *r*) $\overset{\text{def}}{=}$
  LCE(*r*) ⇒ *CPUL*(*r*) ∧ *LINUX_CODE_SIGNED*(*r*)

  If the next CPU instruction execution succeeds and that instruction is located in Linux memory, then the CPU is configured to execute Linux, and that instruction is located in a memory block whose contents have a signature in the golden image. Intuitively, if the CPU executes Linux code, then that Linux code is signed.

All of these functions defined on real states are also defined similarly for ideal states, except for *hvm_to_spec* (which does not need to be defined for ideal states since the spec_state instance of an ideal state *i* is simply accessed as *i.spec*). The only exception is *LINUX_CODE_SIGNED*, which for ideal states is defined as:

bool *LINUX_CODE_SIGNED*(ideal_state *i*) $\overset{\text{def}}{=}$
∀*code* ∈ <word32768>. *code* ∈ *linux_code*(*i*) ⇒ *sign*(*code*) ∈ *i.spec.GI*.

6.2 Definition of Goal and Structure of Proof Plan

This section formally defines the goal, and describes the top-level lemmas and how they are applied to prove the goal.

6.2.1 Formal Definition of Goal

The purpose of the proof plan is to describe how Theorem I can be formally proved:

$$\text{Theorem I.} \qquad \forall r \in RM.S. \quad EXEC\_SIGNED\_LINUX\_CODE(r) \wedge \neg r.nic.dead.$$

For each state that the hardware can reach during a system execution, if Linux code can be executed from that state, then in that state, the CPU is configured to execute Linux and that Linux code is signed, and in addition, the NIC is in a defined state. Apart from the obvious that executed Linux code must be signed, the CPU must always be configured to execute Linux (in non-privileged mode) when it is executing Linux in order for the hypervisor to control the system. The NIC must always be in a defined state since otherwise, it is unknown which operations the NIC performs. If the NIC performs certain memory writes, those writes could lead to that the CPU executes unsigned Linux code or that Linux gets in control of the system. Theorem I is what shall be proved in HOL4 by means of the real model.

6.2.2 Structure of Proof Plan

This subsection introduces the top-level lemmas and provides a birds-eye-view of the proof plan by showing how these lemmas are applied to prove Theorem I. The proof plan consists of four parts. Part 1 is concerned with proving that in the ideal model, the executed binary code of Linux is located in blocks whose contents are signed. Part 2 is concerned with proving that the executions of the binary code of Linux is identical on the binary interfaces described by the ideal model and the real model. This proof is constructed by applying the simulation proof method on the ideal model and the real model. Part 3 is concerned with proving the final three top-level lemmas that are used to transfer the property of only signed Linux code execution from the ideal model to the real model, and which are derived from the first six top-level lemmas proved in parts 2 and 3. Part 4 is concerned with proving Theorem I by applying the final three top-level lemmas proved in Part 3. A bird-eye-view of these four parts is given by the following four subsections.

6.2.2.1 Part 1: Formal Software Design Is Correct

The formal software design of the hypervisor and the monitor is correct if, in the ideal model, the executed binary code of Linux is signed. To prove that the executed binary code of Linux is signed, the following four steps can be taken.

First, a security invariant, *SEC*, on ideal states is defined. The purpose of *SEC* is twofold:

- Imply Theorem I for ideal states: If an ideal state satisfies *SEC*, then for that state (i) if Linux is executed, then the CPU is configured to execute Linux and signed code is executed, and (ii) the NIC is in a defined state.
- Restrict a satisfying state such that no transition from that state can falsify *SEC*.

Second, the first purpose of *SEC* is proved, giving Lemma I:

$$\text{Lemma I.} \quad \forall i \in S_{ideal}. \; SEC(i) \Rightarrow EXEC\_SIGNED\_LINUX\_CODE(i) \land \neg i.nic.dead.$$

Third, all initial states of the ideal model are defined to satisfy *SEC*:

$$\forall i \in IM.IS.\ SEC(i).$$

This is done when the ideal model is defined (see the definitions of *IM.IS* and IS_{ideal} in Subsection 5.3.3).

Fourth, the second purpose of *SEC* is proved by proving that *SEC* is preserved by:

- CPU transitions starting from states in which the CPU is in non-privileged mode: These transitions describe executions of CPU instructions located in Linux memory. Since Linux is untrusted, any CPU instruction can potentially be executed, and therefore all CPU instructions must be considered.
- Specification transitions starting from states in which the CPU is in privileged mode: These transitions describe the operations of the formal software design of the hypervisor and the monitor. In contrast to the previous case, these operations are known.
- NIC transitions: These transitions describe the operations of the NIC.

These three properties are possible to prove because of the restrictions *SEC* impose on the states satisfying *SEC*. The three proofs imply that *SEC* is preserved by all transitions in the ideal model (cf. the upper part of Figure 12 in Subsection 2.3.4.2):

$$\forall i \rightarrow_I i' \in IM.\delta.\ SEC(i) \Rightarrow SEC(i').$$

Steps three and four imply that all states in the ideal model satisfy *SEC*, giving Lemma II:

Lemma II. $\forall i \in IM.S.\ SEC(i).$

Lemma I and Lemma II imply Lemma III:

Lemma III. $\forall i \in IM.S.$
$EXEC\_SIGNED\_LINUX\_CODE(i) \land \neg i.nic.dead.$

Lemma III states that for all states in the ideal model, only signed Linux code can be executed and the NIC is in a defined state. That is, the formal software design of the hypervisor and the monitor is correct. As motivated in Section 5.3 and mentioned in the opening of this chapter, once the formal software design is proved to be correct, the formal software design can be considered as the formal specification of the implementation of the hypervisor and the monitor. As can be seen, Lemma III is Theorem I but with respect to the ideal model instead of the real model. It is the property of Lemma III that shall be transferred to the real model by means of Lemma VII, VIII and IX, proved in part 3. To be able to transfer the property of Lemma III, it must first be proved that the executions of the binary code of Linux are identical in the ideal model and in the real model. This is done in part 2 by means of the simulation proof method.

6.2.2.2 Part 2: Implementation of Hypervisor and Monitor Is Correct

The implementation of the formal software design of the hypervisor and the monitor is correct, if the executions of the binary code of Linux are identical in the ideal model and in the real model (since only signed Linux code is executed in the ideal model). To prove that the executions of the binary code of Linux are identical in the ideal model and in the real model, the following four steps can be taken.

First, to prove that the executions of the binary code of Linux are identical in the ideal model and in the real model by means of the simulation proof method, a simulation relation $R \subseteq S_{real} \times S_{ideal}$ is defined. $(r, i) \in R$ is written as $r\ R\ i$. For R to be useful, for $r \in RM.S$ and $i \in IM.S$, $r\ R\ i$ must imply the following three properties:

- Lemma III can be transferred from *i* to *r*.
- For each transition starting from *r*, and describing an execution of a CPU instruction located in Linux memory or a fine-grained NIC operation, there exists a transition starting from *i*, such that the two transitions describe identical operations.

- For each sequence of transitions starting from r, and describing an execution of an exception handler of the hypervisor and the monitor, there exists a sequence of transitions starting from i, such that the two complete transition sequences describe identical operations.

For $r\ R\ i$ to imply these three properties, R must require equality between the state components of r and i that affect either Lemma III or the set of possible transitions starting from r or i (the state components that the transition rules in Subsection 5.3.2 depend on):

- State components affecting the execution of Linux: The CPU register state components, and the memory state component entries corresponding to the memory region allocated to Linux. These CPU and memory state components exist both in real states and in ideal states.
- State components affecting the execution of the hypervisor and the monitor and the operations specified by the formal specification (software design): The memory state component entries corresponding to the memory regions allocated to the hypervisor and the monitor of the real state, and the *spec* state component of the ideal state.
- State components affecting the operation of the NIC: The *nic* state component of the real state and the ideal state.

Figure 39 gives a graphical illustration of which state components of a real state and an ideal state that must be equal in order for the two states to be related by R. A formal definition of R is given in Subsection 6.4.1.

Second, in order for the application of the simulation proof method on RM, IM with respect to R, to enable the transfer of Lemma III from a real state to an ideal state, two additional labeled transition systems, the real Linux model and the ideal Linux model, are defined in terms of the real model and the ideal model, respectively. The real Linux model and the ideal Linux model are denoted by the four tuples RLM and ILM, respectively:

- $RLM = (\{r \mid r \in RM.S \land CPUL(r)\}, RM.IS, \delta, \Pi)$.
- $ILM = (\{i \mid i \in IM.S \land CPUL(i)\}, IM.IS, \delta, \Pi)$.

Each component of RLM and ILM is (informally) defined in terms of RM and IM as follows:

- The sets of states, $RLM.S$ and $ILM.S$, contain all and only the states in $RM.S$ and $IM.S$, respectively, in which the CPU is configured to execute Linux, as determined by $CPUL$.
- The sets of initial states, $RLM.IS$ and $ILM.IS$, are equal to the sets of initial states in $RM.IS$ and $IM.IS$, respectively.
- The sets of transitions, $RLM.\delta$ and $ILM.\delta$, contain a transition from a state s to a state s', written as $s \rightsquigarrow_{real} s'$ and $s \rightsquigarrow_{ideal} s'$ (the labels of all transitions in $RLM.\delta$ and $ILM.\delta$ are *real* and *ideal*), if and only if there exists an execution trace in $RM.\Pi$ and $IM.\Pi$, respectively, such that in that execution trace there either exists:

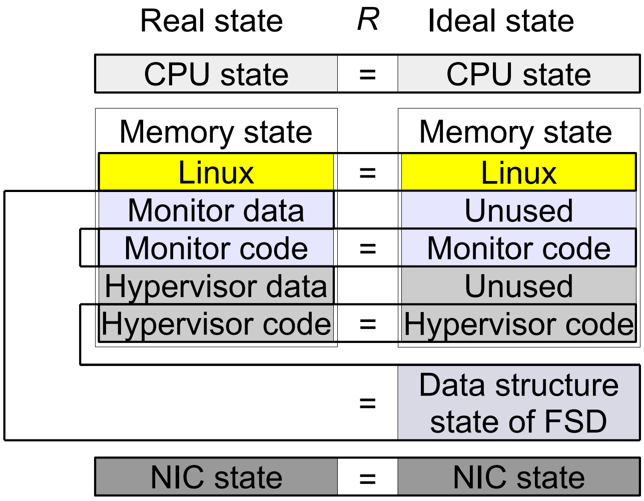


Figure 39: The equality requirements on a real state and an ideal state in order for them to be related by R. A real state has three state components, as shown on the left (and in Figure 36): CPU registers, memory, and NIC. An ideal state has four state components, as shown on the right (and in Subsection 5.3.1), extending a real state with the spec state component holding the values of the data structures of the formal software design (denoted FSD). For a real state and an ideal state to be related by R, the following equalities must hold between the real state and the ideal state: (i) the CPU state components, (ii) the Linux memory entries of the memory state components, (iii) the NIC state components, (iv) the hypervisor and monitor memory entries corresponding to where the code of the hypervisor and the monitor is stored, and (v) the values of the data structures of the hypervisor and the monitor stored in the hypervisor and monitor memory entries of the memory state component of the real state, must be equal to the values of the corresponding data structures stored in the spec state component of the ideal state. The hypervisor and monitor memory entries of the memory state component of the ideal state do not affect transitions in the ideal model. The proof of Lemma V relies on sub-level lemmas that depend on the equality of the hypervisor and monitor code. The hypervisor and monitor memory entries corresponding to where the data of the hypervisor and the monitor are stored are irrelevant in the ideal model.

- ○ a transition from s to s', describing the execution of a CPU instruction located in Linux memory,
- ○ a sequence of transitions starting from s and ending in s', consisting of (i) transitions describing an execution of an exception handler, and (ii) the longest consecutive sequence of NIC transitions immediately following the exception handler execution transitions, or
- ○ a NIC transition from s to s', not a occurring in a sequence of transitions of the type described in the previous item bullet.

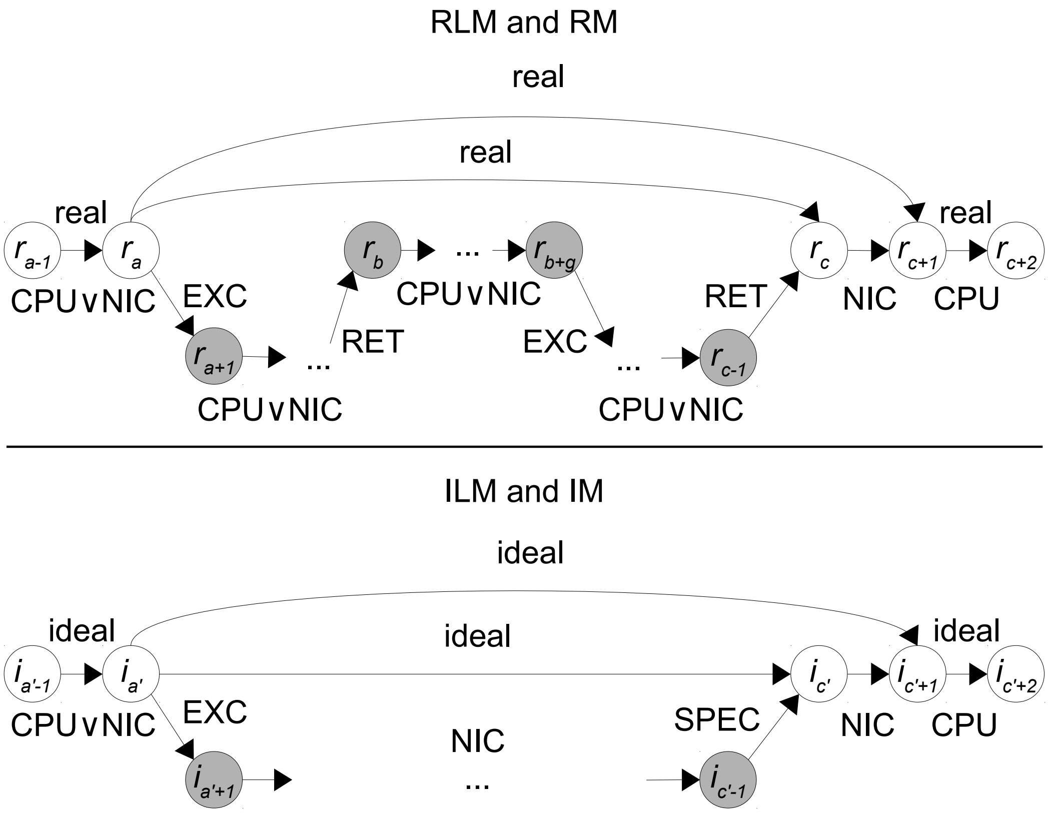


Figure 40: The relationship between RLM and RM, and ILM and IM. The upper part of the figure shows sequences of transitions in RLM.δ and RM.δ, while the lower part of the figure shows sequences of transitions in ILM.Π and IM.Π. White states satisfy CPUL and shaded states do not satisfy CPUL. RLM.S and ILM.S include only the white states, while RM.S and IM.S include all states. RLM.δ and ILM.δ include only transitions with the label real or ideal, respectively, while RM.δ and IM.δ include transitions with the other labels. A transition with the label CPU∨NIC is either a CPU transition or a NIC transition. The transitions from r_{a-1} to r_a and from r_{c+1} to r_{c+2} are both in RM.δ and RLM.δ, but in RLM.δ they are denoted by $r_{a-1} \rightsquigarrow_{real} r_a$ and $r_{c+1} \rightsquigarrow_{real} r_{c+2}$, respectively. Similarly for the transitions from $i_{a'-1}$ to $i_{a'}$, and from $i_{c'+1}$ to $i_{c'+2}$, denoted by $i_{a'-1} \rightsquigarrow_{ideal} i_{a'}$ and $i_{c'+1} \rightsquigarrow_{ideal} i_{c'+2}$ in ILM.δ, respectively. CPU transitions describing the execution of Linux can only occur from white states. The CPU transitions between r_{a+1} and r_b and between r_{b+g+1} and r_c describe executions of the hypervisor, which are possibly interleaved with NIC transitions. The CPU transitions between r_b to r_{b+g+1} describe an execution of the monitor, also possibly interleaved with NIC transitions. With respect to an execution trace in RM.Π or IM.Π, what RLM and ILM exclude but what RM and IM include are (i) the states and transitions that occur in transition sequences describing executions of exception handlers, and (ii) the NIC transitions immediately following transition sequences describing executions of exception handlers. $r_{a-1} \rightsquigarrow_{real} r_a$ is either a Linux transition or a NIC transition. $r_a \rightsquigarrow_{real} r_c$ and $r_a \rightsquigarrow_{real} r_{c+1}$ are exception handler transitions, defined in terms of two different execution traces in RM.Π (in the execution trace defining $r_a \rightsquigarrow_{real} r_c$, r_c is followed by a CPU transition). There is no transition from r_c to r_{c+1} in RLM.δ because it is a NIC transition following a transition sequence of an exception handler execution.

This gives three types of transitions in *RLM.δ* and *ILM.δ*: Linux transitions, exception handler transitions and NIC transitions, respectively. An intuitive explanation of why certain transitions and states in *RM* and *IM* are excluded from *RLM* and *ILM*, respectively, is given below in this subsection and in Subsection 6.4.2.

- The sets of execution traces, *RLM.Π* and *ILM.Π*, contain an execution trace if and only if that execution trace starts from an initial state in *RLM.IS* or *ILM.S* and consists of a sequence of transitions in *RLM.δ* or *ILM.δ* of arbitrary length, respectively.

Figure 40 illustrates the relationship between *RLM* and *RM*, and *ILM* and *IM*. Previously, it was stated that the simulation proof method was applied on *RM* and *IM*. Actually, the simulation proof method is applied on *RLM* and *ILM*, but which are defined in terms of *RM* and *IM*, respectively. *RLM* and *ILM* are formally defined in Subsection 6.4.2.

Third, it is proved that each initial state in *RLM.IS*, is related by *R* to some initial state in *ILM.IS*, giving Lemma IV:

Lemma IV. $\forall r \in RLM.IS.\ \exists i \in ILM.IS.\ r\ R\ i.$

Fourth, the simulation proof method is applied to prove that each transition in *RLM.δ* has a matching transition in *ILM.δ* with respect to *R*, giving Lemma V:

Lemma V. $\forall r, r' \in RLM.S,\ i \in ILM.S.$
$r \rightsquigarrow_{real} r' \wedge r\ R\ i \Rightarrow \exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \wedge r'\ R\ i'.$

Figure 41 gives a graphical illustration of Lemma V. The meaning of Lemma V is that, for each real state in *RM.S*, from which Linux code can be executed (*CPUL* holds), there exists an ideal state in *IM.S*, such that (i) the state of Linux is identical in the two states, and (ii) the operations performed from the two states are identical and affect the execution of Linux identically (with respect to the binary interface: execution of a Linux instruction, execution of an exception handler, or an execution step of the NIC).

Lemma V can be proved by proving it separately for the three types of transitions in *RLM.δ* (Linux, exception handler and NIC transitions). Lemma V can be proved for Linux transitions ($r_c \rightsquigarrow_{real} r_{c+1}$ in Figure 41) since *R* requires the CPU state components, the Linux memory region entries of the memory state components, and the NIC state components to be equal (recall that the CPU is allowed to read NIC registers although the CPU is executing Linux). This means that the CPU will execute identical CPU instructions with identical operands from related states. The CPU will therefore perform identical operations from related states when executing Linux.

Proving Lemma V for exception handler transitions ($r_a \rightsquigarrow_{real} r_c$ in Figure 41) is more problematic. The reason is that in *RM.Π*, NIC transitions can occur between CPU transitions (including exception transitions occurring due to monitor executions and exception return transitions, having labels *EXC* and *RET*, respectively) describing executions of exception handlers, while in *IM.Π*, this is

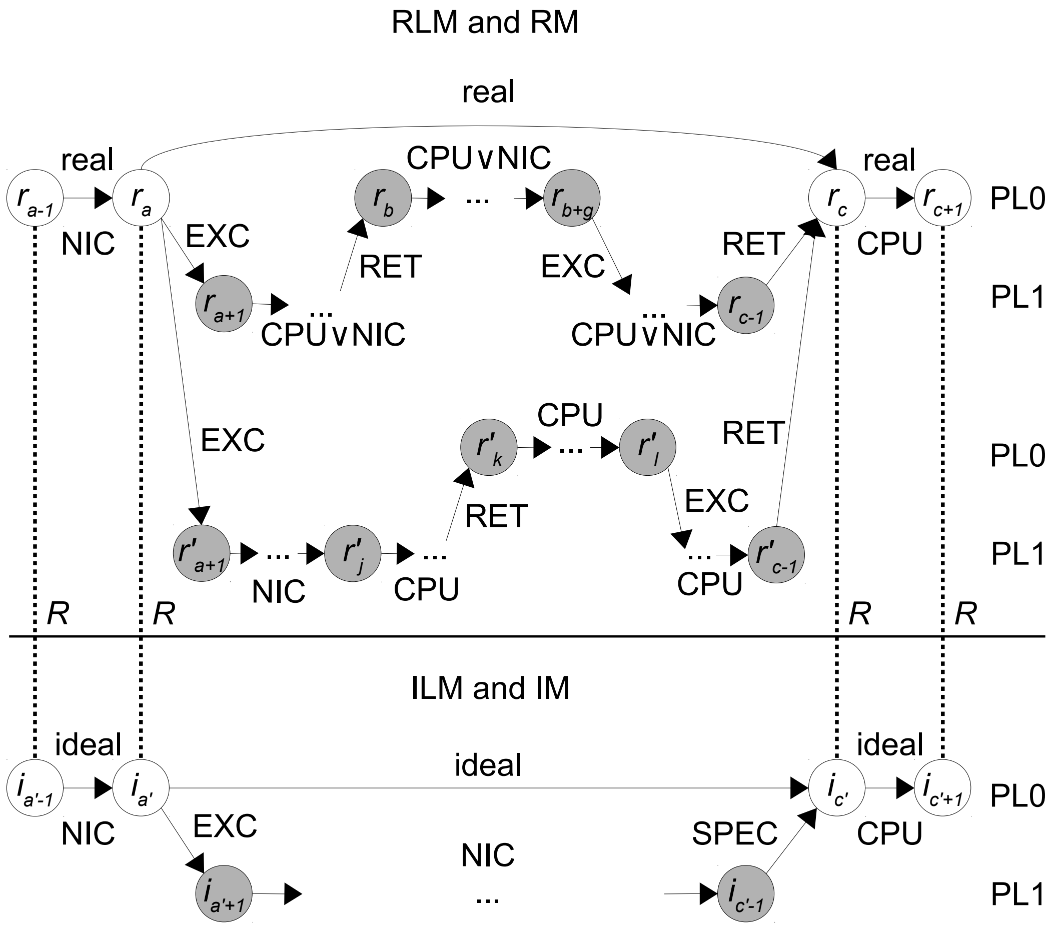


Figure 41: The main components involved in Lemma V. The upper half of the figure shows one sub-execution trace in RLM.Π and two sub-execution traces in RM.Π. The lower part of the figure shows one sub-execution trace in ILM.Π and one sub-execution trace in IM.Π. States related by R are connected by dashed lines having the label R. For each of the transitions $r_{a-1} \rightsquigarrow_{real} r_a$, $r_a \rightsquigarrow_{real} r_c$ and $r_c \rightsquigarrow_{real} r_{c+1}$ there exists a matching transition $i_{a'-1} \rightsquigarrow_{ideal} i_{a'}$, $i_{a'} \rightsquigarrow_{ideal} i_{c'}$ and $i_{c'} \rightsquigarrow_{ideal} i_{c'+1}$ with respect to R, respectively. Since r_{a-1} R $i_{a'-1}$ holds, this implies that r_a R $i_{a'}$, r_c R $i_{c'}$ and r_{c+1} R $i_{c'+1}$ hold.

not the case since the operations of the exception handlers are specified by atomic specification transitions. The problem is to prove that the CPU transitions in *RM.Π* combined (between r_a and r_c in the upper sub-execution trace in Figure 41) describe identical operations as described by the corresponding specification transition in *IM.Π* ($i_{c'-1} \rightarrow_{SPEC} i_{c'}$). The same problem applies to the interleaved NIC transitions in *RM.Π* (between r_a and r_c in the upper sub-execution trace in Figure 41) and the corresponding NIC transitions in *IM.Π* (between $i_{a'}$ and $i_{c'}$). These two problems can be solved by finding another scheduling of CPU and NIC transitions in *RM.Π* such that:

- the CPU transitions occur in consecutive sequence (between r'_j and r_c) with the NIC transitions occurring before or after those CPU transitions (between r'_{a+1} and r'_j), and
- the rescheduled sub-execution trace (between r_a and r_c in the lower sub-execution trace in Figure 41) describe identical operations as described by the original sub-execution trace (between r_a and r_c in the upper sub-execution trace in Figure 41).

Important is that the original exception handler sub-execution trace and the rescheduled one start from the same state and end in the same state (r_a and r_c, respectively).

Lemma V can then be proved for exception handler transitions in four steps by traversing transition by transition in turn and order in the rescheduled sub-execution trace (between r_a and r_c in the lower sub-execution trace in Figure 41) and the corresponding sub-execution trace in *IM.Π* (between $i_{a'}$ and $i_{c'}$). First, it is proved that if a CPU exception occurs from a real state ($r_a \rightarrow_{EXC} r'_{a+1}$), and that real state is related by *R* to an ideal state in *IM.S* ($i_{a'}$), then the same CPU exception occurs from the ideal state ($i_{a'} \rightarrow_{EXC} i_{a'+1}$) and the same operations are performed by the CPU exceptions. Second, it is proved that identical operations are described by each pair of NIC transitions in the rescheduled sub-execution trace and in the corresponding sub-execution trace in *IM.Π* before the CPU and specification transitions (between r'_{a+1} and r'_j, and between $i_{a'+1}$ and $i_{c'-1}$), respectively. Third, it is proved that all CPU transitions in the rescheduled sub-execution trace combined (between r'_j and r_c) describe identical operations as described by the corresponding specification transition in the sub-execution trace in *IM.Π* ($i_{c'-1} \rightarrow_{SPEC} i_{c'}$). Fourth, if NIC transitions occur after the CPU transitions in the rescheduled trace (not shown in Figure 41), then the first step is repeated. These four steps imply that *R* is preserved by: initial exception transitions, NIC transitions, and by corresponding CPU and specification transitions. The four steps are described formally in Subsection 6.4.4.3.

Lemma V can be proved for exception handler transitions in *RLM.δ* since *R* requires equality between (i) the Linux states, (ii) the hypervisor and monitor memory entries in the real state, and the *spec* state component and hypervisor and monitor code memory entries in the ideal state, and (iii) the *nic* state components. The first equality requirement implies identical operations of initial exceptions. The second and third equality requirements imply identical operations of exception handlers executions (which might access NIC registers potentially causing exception handler executions and NIC operations to affect each other) and the NIC.

Lemma V can be proved for NIC transitions ($r_{a-1} \rightsquigarrow_{real} r_a$ in Figure 41) since *R* requires the *nic* state components to be equal. The operations described by NIC transitions depend only on the state of the NIC. The NIC performs therefore identical operations from related states.

It can now be understood why the simulation proof method is applied on *RLM* and *ILM* and not directly on *RM* and *IM*. It is not possible to prove Lemma V with respect to *RM.S* and *IM.S*. The reason is that, in general, intermediate states in executions of exception handlers in *RM.Π* are not related to any state in *RM.S* due to the atomic execution of exception handlers in *IM.Π* (this is further explained in Subsection 6.4.2). Also, to prove that only signed Linux code is executed, it is sufficient to only include states from which Linux code *can* be executed. *RLM.S* and *ILM.S* therefore only include states in *RM.S* and *IM.S*, respectively, from which Linux code *can* be executed (states satisfying *CPUL*). Similarly, *RLM.δ* and *ILM.δ* therefore only include transitions between such states.

Hence, the execution traces in *RLM.Π* and *ILM.Π* include and describe the same states and operations as included and described by the execution traces in *RM.Π* and *IM.Π*, respectively, except in two respects. First, intermediate states in exception handler executions are omitted from *RM.S* and *IM.S*. Second, individual transitions occurring during exception handler executions and NIC transitions occurring immediately after exception handler executions are omitted from *RLM.Π* and *ILM.Π*. That is, *RLM.Π* and *ILM.Π* describe execution traces in which executions of exception handlers are atomic. Considering the execution of Linux, these two omissions from *RLM* and *ILM* of states and transitions with respect to *RM* and *IM*, respectively, do not omit any states from which Linux *can* be executed, nor the granularity of the operations affecting the execution of Linux. *RLM* and *ILM* can therefore be used to prove that the binary executions of Linux in *RM* and *IM* are equal. In other words, *RLM* and *ILM* can be used to prove that the implementation of the hypervisor and the monitor is correct.

Lemma IV and Lemma V imply that each state in *RLM.S* is related by *R* to some state in *ILM.S*, giving Lemma VI:

Lemma VI. $\forall r \in RLM.S.\ \exists i \in ILM.S.\ r\ R\ i.$

6.2.2.3 Part 3: Three Lemmas Transferring Lemma III to RM

The final three top-level lemmas, depending on the first six top-level lemmas, are used to transfer Lemma III from *IM* to *RM*.

First, Lemma VII states two properties with respect to *RM*:

- If the CPU is executing Linux, then the CPU is configured to execute Linux.
- If the CPU is not configured to execute Linux, then the NIC is in a defined state.

Lemma VII. $\forall r \in RM.S.$
$$[LCE(r) \Rightarrow CPUL(r)] \wedge [\neg CPUL(r) \Rightarrow \neg r.nic.dead].$$

As mentioned in Subsection 2.1.1, brackets are used as ordinary parenthesis to ease the interpretation of their scope.

Second, Lemma VIII states that for a real state in *RM.S*, if the CPU is configured to execute Linux, then that real state is related by *R* to some ideal state in *IM.S*:

Lemma VIII. $\forall r \in RM.S.\ CPUL(r) \Rightarrow \exists i \in IM.S.\ r\ R\ i.$

Third, Lemma IX states that if a state in *RM.S* is related by *R* to some state in *IM.S*, then, with respect to that state in *RM.S*, if the CPU will execute a Linux instruction, then that Linux instruction is located in a signed code block, and the NIC is in a defined state:

$$\text{Lemma IX.} \quad \begin{array}{l} \forall r \in RM.S,\ i \in IM.S. \\ r\ R\ i \\ \Rightarrow \\ [LCE(r) \Rightarrow LINUX\_\ CODE\_\ SIGNED(r)] \wedge \\ \neg r.nic.dead. \end{array}$$

6.2.2.4 Part 4: Proof of Theorem I

Proving Theorem I,

$$\forall r \in RM.S.\ EXEC\_SIGNED\_LINUX\_CODE(r) \wedge \neg r.nic.dead,$$

is done in two steps by proving the two conjuncts separately. Consider first *EXEC_SIGNED_LINUX_CODE*(*r*) which is defined to be equal to

$$LCE(r) \Rightarrow CPUL(r) \wedge LINUX\_CODE\_SIGNED(r).$$

Assume *LCE*(*r*) holds for some $r \in RM.S$. Applying Lemma VII, VIII and IX in sequence gives:

1. By Lemma VII, *CPUL*(*r*) holds.
2. By Lemma VIII, $\exists i \in IM.S.\ r\ R\ i$ holds.
3. By Lemma IX, $LCE(r) \Rightarrow LINUX\_CODE\_SIGNED(r)$ holds.

Since *LCE*(*r*) is assumed, *LINUX_CODE_SIGNED*(*r*) holds. Hence,

$$CPUL(r) \wedge LINUX\_CODE\_SIGNED(r)$$

holds.

Consider the second conjunct of Theorem I, $\neg r.nic.dead$. Lemma VII gives

$$\neg CPUL(r) \Rightarrow \neg r.nic.dead,$$

and Lemma VIII and IX give

$$CPUL(r) \Rightarrow \neg r.nic.dead.$$

Irrespectively of whether *r* satisfies *CPUL* or not, $\neg r.nic.dead$ holds.

Since $r \in RM.S$ was arbitrarily chosen,

$$EXEC\_SIGNED\_LINUX\_CODE(r) \wedge \neg r.nic.dead$$

holds for all states in *RM.S*.

Figure 42 gives an overview of the proof plan by showing the dependences between Theorem I, the top-level lemmas, the sub-level lemmas, and some assumptions made in the proof plan. The first three parts of the proof plan, sketched in the previous Subsections 6.2.2.1 through 6.2.2.3, are described in deeper detail in the following three sections. Those following three sections describe how the top-level lemmas can be proved.

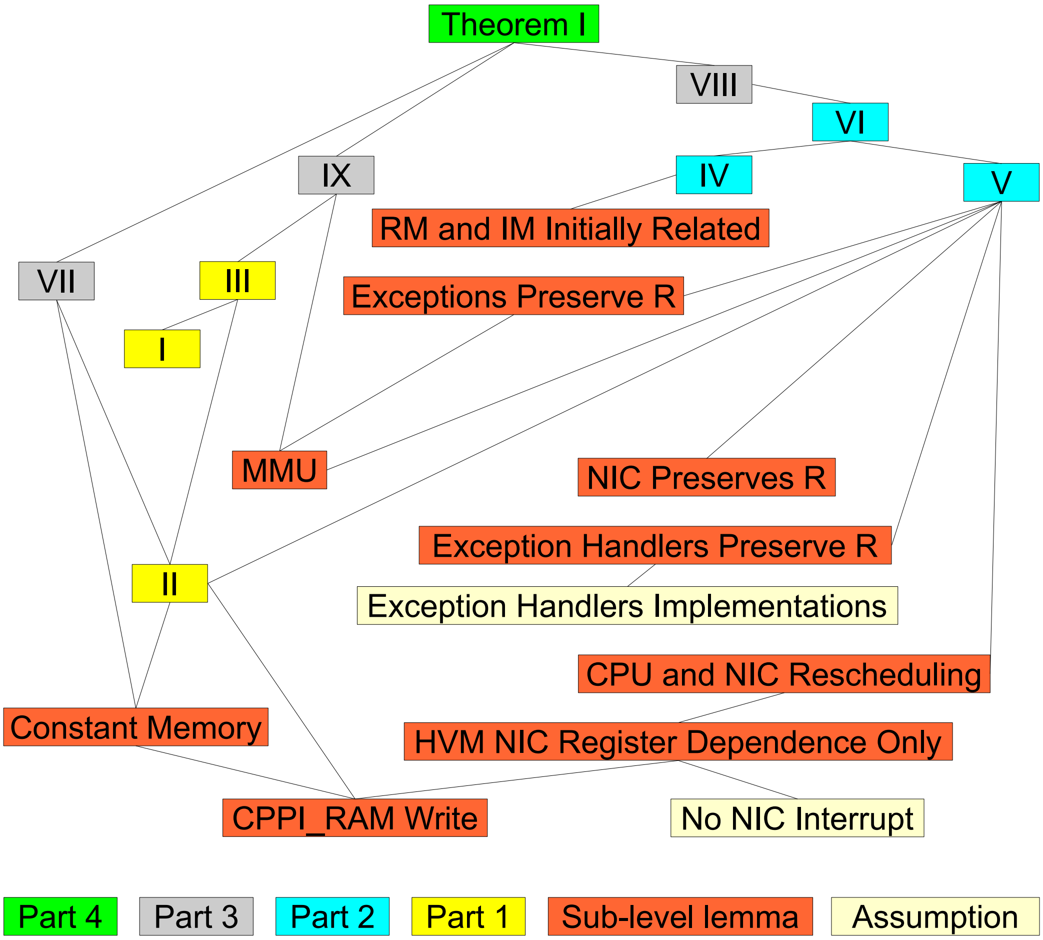


Figure 42: The dependences between Theorem I, the top-level lemmas, the sub-level lemmas, and the assumptions made in the proof plan. The boxes with roman numerals correspond to the top-level lemmas with the same numerals. The other boxes, except the one with the label Theorem I, correspond to the sub-level lemmas or the assumptions with the same name. A statement depends on another statement if the box of the former statement is placed above the box of the latter statement and the boxes are connected by a line. For instance, Lemma VI depends on Lemma IV and V, and Lemma IV depends on (the sub-level lemma) RM and IM Initially Related Lemma. The sub-level lemmas are described and motivated in Appendix F, where also the assumptions are described.

6.3 Lemma III: Formal Software Design Is Correct

The purpose of this section is to describe the details of the first part of the proof plan and how Lemma III can be proved. The presentation of this section follows the steps outlined in Subsection 6.2.2.1, except for the omission of step three of defining *IM.IS* such that all ideal states in *IM.IS* satisfies the security invariant *SEC* which is done in Section 5.3.3. Subsection 6.3.1 describes the parts of *SEC* that are involved in reasonings presented later in this chapter. Subsection 6.3.2 is devoted to Lemma I where it is described how it can be proved that *SEC* implies that only Linux code is executed from the ideal state satisfying *SEC* and that the NIC state is defined in that ideal state, thereby proving Lemma I. Subsection 6.3.3 is devoted to Lemma II where it is described how it can be proved that all transitions in the ideal model, *IM.δ*, preserve *SEC*. Finally, a simple motivation is given of why Lemma I and Lemma II imply Lemma III. (All reasonings in this section are with respect to the ideal model *IM*.)

6.3.1 Security Invariant SEC

The security invariant *SEC* must be defined with Lemma I and Lemma II in consideration:

- Lemma I: The properties an ideal state must have in order to imply that only signed Linux code is executed from that ideal state and that the NIC is in a defined state in that ideal state.
- Lemma II: The properties an ideal state must have such that the next transition from it does not cause the succeeding state to falsify *SEC*.

These two considerations together with the development of the proof plan have led to the following definition of *SEC*:

$$\mathit{SEC}(\text{ideal\_state } i) \overset{\text{def}}{=} \mathit{CPU\_MEMORY}(i) \land \mathit{NIC}(i),$$

where

$$\mathit{CPU\_MEMORY}(\text{ideal\_state } i) \overset{\text{def}}{=} \mathit{WT\_EX\_REF}(i) \land \mathit{SOUND\_PT}(i) \land \mathit{CONST\_PT}(i) \land \mathit{SOUND\_MMU}(i) \land \mathit{LINUX}(i),$$

$$\mathit{NIC}(\text{ideal\_state } i) \overset{\text{def}}{=} \mathit{FINITE\_WORD\_ALIGNED\_CPPI\_RAM\_QUEUES}(i) \land \mathit{NIC\_BDS}(i) \land \mathit{NO\_BD\_OVERLAPS}(i) \land \mathit{NIC\_DATA\_NO\_EXEC\_CONF}(i) \land \mathit{NIC\_READ\_ONLY}(i) \land \mathit{CANNOT\_DIE}(i) \land \mathit{TD\_STOP\_NIC}(i) \land \mathit{RECV\_BD\_REF}(i) \land \mathit{INIT\_TD\_IDLE}(i) \land \mathit{RX\_BUFFER\_OFFSET\_DMACONTROL\_ZERO}(i) \land \mathit{ACTIVE\_CPPI\_RAM}(i).$$

Each predicate is described and formally defined in Appendix E. Due to space limitations, this subsection gives intuitive descriptions of only four predicates of *CPU_MEMORY* and two predicates of *NIC*. Those descriptions are enough to understand the lemmas and definitions described in this chapter.

The purpose of the predicate *CPU_MEMORY* is to ensure that the CPU, MMU, memory and certain data structures of the formal software design are in consistent and secure states (with respect to ensuring that executed Linux code is signed). This subsection describes the following four predicates of *CPU_MEMORY*:

- *SOUND_PT*:

  All page table entries in L1 and L2 blocks are secure: (i) Blocks are not mapped by page tables in L1 and L2 blocks as both writable and executable, and if blocks are executable then their contents are signed, and (ii) all second-level entries of page tables in L1 blocks refer to page tables in L2 blocks.

SOUND_PT is used to prevent unsigned Linux code from being executed by the CPU.

- *CONST_PT*:

  Execution of Linux cannot cause the CPU to write L1 or L2 blocks.

  CONST_PT prevents the execution of Linux code from changing: (i) the access permissions to enable the CPU to write and then execute unsigned Linux code, and (ii) the memory mapping to enable access to hypervisor or monitor memory.

- *SOUND_MMU*:

  The first-level page table that the MMU uses in an address translation is in a block of type L1. Also, the virtual to physical address mappings of the hypervisor and the monitor are as specified (according to the compilation of their source code), and the memory regions of the hypervisor and the monitor contain their specified code (produced by the compilation of their source code).

  SOUND_MMU ensures that when an exception occur, the CPU executes hypervisor code, and together with *SOUND_PT*, *SOUND_MMU* ensures that the MMU applies the security policy (the CPU can only execute signed Linux code). The former property is used to prove Lemma V.

- *LINUX*:

  If the CPU is executing Linux, then the CPU is configured to do so, and τ is correct:

  ○ *If the program counter is mapped by the MMU to a physical address allocated to Linux, then the CPU is in non-privileged mode with DACR[5:4] = 0b01.*

  ○ *Blocks typed as L1, L2 or D are allocated to Linux.*

  ○ *Blocks typed as ⊥ are either unmapped by page tables in L1 and L2 blocks, or mapped by page tables in L1 and L2 blocks as inaccessible to the CPU when the CPU executes Linux.*

  ○ *Blocks containing NIC registers that affect which memory accesses the NIC performs are typed as MN.*

  ○ *The block that contains NIC registers, none of which affect which memory accesses the NIC performs, is typed as N.*

  ○ *Blocks allocated to the hypervisor or the monitor are typed as ⊥.*

  LINUX ensures that Linux is only executed when the CPU is in non-privileged mode with secure access permissions. It also allows hypervisor and monitor memory to be securely mapped by page tables in *L1* and *L2* blocks, enabling the proof of Lemma V.

The purpose of the predicate *NIC* is to ensure that: (i) the NIC does not enable the CPU to execute unsigned Linux code, (ii) the NIC is in a defined state, and (iii) the values of the data structures of the formal software design correctly reflect the state of the NIC. This subsection describes the following two predicates of *NIC*:

- *NIC_READ_ONLY*:

  When the CPU is configured to execute Linux, the CPU cannot: (i) execute CPU instructions encoded by the contents of NIC registers, (ii) write NIC registers that affect which memory accesses the NIC performs, nor (iii) perform accesses to unaligned physical addresses, at which NIC registers are located.

  NIC_READ_ONLY prevents the CPU, when configured to execute Linux, from executing unsigned code by fetching the contents of NIC registers, decode the fetched contents as instructions, and then execute the decoded instructions. Also, *NIC_READ_ONLY* prevents the CPU when executing Linux from reconfiguring the NIC to enter an insecure state, or access NIC registers at unaligned physical addresses causing the NIC to enter an undefined state.

- *CANNOT_DIE*:

  The NIC cannot enter an undefined state when executing from its current state.

  CANNOT_DIE ensures that the NIC can only enter NIC states that are defined when performing an arbitrary number of autonomous transitions, including zero, from the current NIC state of the ideal state.

6.3.2 Lemma I: SEC Is Secure

This subsection describes how it can be proved that *SEC* implies (i) that when the CPU executes Linux, the CPU executes signed code in non-privileged mode, and (ii) that the NIC is in a defined state. That is a description of how Lemma I can be proved:

$$\forall i \in S_{ideal}.\ SEC(i) \Rightarrow EXEC\_SIGNED\_LINUX\_CODE(i) \wedge \neg i.nic.dead.$$

Assume *SEC*(*i*) holds for $i \in S_{ideal}$. The predicate *CANNOT_DIE* of *SEC* implies that the NIC is not in an undefined state in the ideal state *i*. The other conjunct in the consequent of the implication, *EXEC_SIGNED_LINUX_CODE*(*i*), is defined as

$$LCE(i) \Rightarrow CPUL(i) \wedge LINUX\_CODE\_SIGNED(i).$$

Assume *LCE*(*i*). Consider first *CPUL*(*i*). The first bullet item in the description of the predicate *LINUX* in the previous Subsection 6.3.1 states that *LINUX* includes the predicate

"*If the program counter is mapped by the MMU to a physical address allocated to Linux, then the CPU is in non-privileged mode with DACR[5:4] = 0b01.*"

According to the definitions of *LCE* and *CPUL* in Section 6.1, this an informal formulation of

$$LCE(i) \Rightarrow CPUL(i).$$

Since *LCE*(*i*) is assumed, this implication means that *CPUL*(*i*) holds.

Consider *LINUX_CODE_SIGNED*(*i*), which is defined as

$$\forall code \in \text{<word32768>}.\ code \in linux\_code(i) \Rightarrow sign(code) \in i.spec.GI.$$

Assume that a bit string of length 32768 bits (4 kB), *code*, is in *linux_code*(*i*). By the definition of *linux_code*, *code* is contained in a memory block allocated to Linux and mapped by the MMU as executable. Since *SEC*(*i*) implies *SOUND_MMU*(*i*) and *SOUND_PT*(*i*), the MMU computes address mappings via page tables located in *L1* and *L2* blocks. In addition, *SOUND_PT*(*i*) states that if page tables in *L1* and *L2* blocks maps blocks as executable, then their contents are signed. Hence, the signature of *code* is in the golden image.

6.3.3 Lemma II: Ideal Model Satisfies SEC

This subsection describes how it can be proved that all states in the ideal model satisfy *SEC*. That is a description of how Lemma II can be proved:

$$\forall i \in IM.S.\ SEC(i).$$

In the third step in the first part of the proof plan, described in Subsection 6.2.2.1, it was stated that the ideal model is defined such that all of its initial states satisfy *SEC*:

$$\forall i \in IM.IS.\ SEC(i).$$

It is therefore sufficient to prove Lemma II by proving that all transitions in the ideal model preserve *SEC*, which is the fourth step of the first part of the proof plan:

$$\forall i \rightarrow_l i' \in IM.\delta.\ SEC(i) \Rightarrow SEC(i').$$

This preservation property is proved by considering how the three types of transitions in the ideal model operates. That is, the operations of the following transitions:

- Non-privileged CPU transitions: Describe the execution of Linux instructions. These transitions are described by the CPU model when it is in non-privileged mode in the pre-state *i*.
- Specification transitions: Take the role of the exception handlers and operate according to the formal software design. These transitions occur when the CPU model is in privileged mode in the pre-state *i*.
- NIC transitions: Describe the operations performed by the NIC and how the NIC handles memory read request replies. Each NIC transitions is one autonomous NIC transition possible followed by a memory read request reply transition, if the autonomous transition issued a memory read request.

It has been reasoned that *SEC* is preserved by non-privileged CPU transitions, the extended memory mapping request handlers (which extended the original memory mapping request handlers [86] to take the operation of the NIC into account) and the NIC register write request handlers, which are a part of the specification transitions, and NIC transitions. To give some insight to the reasonings that can be used to prove that all transitions in the ideal model, *IM.δ*, preserve *SEC*, motivations are given of why the six predicates of *CPU_MEMORY* and *NIC* intuitively described in Subsection 6.3.1 are preserved by non-privileged CPU and NIC transitions. Reasonings of how the other predicates are preserved and how specification transitions preserve *SEC* are omitted for space limitations.

For specification transitions, all such transitions modify *i.cpu.uregs.r15* (the program counter) and *i.cpu.sregs.CPSR*[4:0] (used to set execution mode of the CPU) when returning the CPU to Linux in non-privileged mode. *SEC* depends on both of these state components. Of the other state components that *SEC* depends on, *i.cpu.cp15.TTBR0*, *i.cpu.cp15.DACR*, *i.memory*, *i.nic* and *i.spec*, it is only the memory mapping and NIC register write request handlers that modify these state components. (The memory mapping and NIC register write request handlers modify: *TTBR0* when the first-level page table is switched; *DACR* is not accessed by the specification transitions since there is no monitor in the ideal model and *DACR* is only used to manage memory access permissions for Linux and the monitor, as described in the last paragraph in Subsection 2.3.4.1; *memory* when modifying a page table entry; *nic* when writing a NIC register; and *spec* when modifying a data structure.) Apart from proving that all specification transitions correctly restore *r15* and *CPSR*[4:0], it is sufficient to prove that all specification transitions preserve *SEC* by proving that the memory mapping and NIC register write request handlers preserve *SEC* and that all other operations described by the specification transitions (e.g. interrupt handling) do not access *TTBR0*, *DACR*, *memory*, *nic* and *spec*.

Most of the reasonings in the following two subsections, of that non-privileged CPU and NIC transitions preserve the six predicates of *CPU_MEMORY* and *NIC*, are based on that non-privileged CPU and NIC transitions do not modify the state components that *SEC* depends on. Since these reasonings assume that *SEC* holds for the pre-state *i*, this non-modification property implies that the six predicates also hold for the post-state *i'*.

6.3.3.1 Non-Privileged CPU Transitions Preserve SEC

This subsection motivates why non-privileged CPU transitions in the ideal model, $i \rightarrow_{CPU} i'$, $i \rightarrow_{EXC} i' \in IM.\delta$, preserve *SOUND_PT*, *CONST_PT*, *SOUND_MMU* and *LINUX* of *CPU_MEMORY*, and *NIC_READ_ONLY* and *CANNOT_DIE* of *NIC*. The following list motivates why non-privileged CPU transitions preserve the four predicates of *CPU_MEMORY*:

- *SOUND_PT*(*i'*): By *CONST_PT*(*i*) *L1* and *L2* blocks are non-writable, and by *SOUND_MMU*(*i*) and *SOUND_PT*(*i*) the MMU uses only page tables in *L1* and *L2* blocks. *SOUND_PT*(*i*) also states that page tables in *L1* and *L2* blocks do not map executable blocks as writable and that executable blocks have signed contents. That is, *L1*, *L2* and executable blocks cannot be modified by non-privileged CPU transitions from *i*. Since only specification transitions can modify *i.spec.τ*, the sets of *L1* and *L2* blocks are identical in *i* and *i'*. Hence, in *i'* blocks that are mapped by page tables in *L1* and *L2* blocks are still not mapped as both writable and executable, and if blocks are mapped as executable, their contents are still signed. In addition, all second-level entries of page tables in *L1* blocks still refer to page tables in *L2* blocks. This means that *SOUND_PT*(*i'*) holds.

- *CONST_PT*(*i'*): In Section E.4 it is listed which state components each predicate of *SEC* depends on. That list shows that *CONST_PT* depends only on $i.spec.\rho_{wt}$ and $i.spec.\tau$. Since only specification transitions can modify these two data structures of the formal software design, *CONST_PT*(*i'*) holds.
- *SOUND_MMU*(*i'*): Since *TTBR0* and τ cannot be modified by non-privileged CPU transitions (*TTBR0* can only be accessed in privileged mode), the first-level page table used by the MMU in *i'* is still in a block of type *L1*. This is the first property of *SOUND_MMU*.

  The MMU using a first-level page table in an *L1* block, *SOUND_PT*(*i*) (second-level entries of page tables in *L1* blocks refer only to page tables in *L2* blocks), *CONST_PT*(*i*) (*L1* and *L2* blocks cannot be modified), and that *DACR* is not accessible in non-privileged mode, together imply that the memory mappings performed by the MMU are identical in *i* and *i'*, since the MMU only depend on these state components according to *mmu*. The memory mappings of the hypervisor and the monitor are therefore identical in *i* and *i'*. This is the second property of *SOUND_MMU*.

  LINUX(*i*) states that blocks allocated to the hypervisor and the monitor are typed as ⊥, and that page tables in *L1* and *L2* blocks either do not map such blocks or maps them as inaccessible to the CPU when the CPU executes Linux (which the CPU does when it is in non-privileged mode). This page table property and the MMU only using page tables in *L1* and *L2* blocks (as motivated in the reasoning of that *SOUND_PT*(*i'*) holds), imply that the blocks containing the code of the hypervisor and the monitor is unmodified by non-privileged CPU transitions. The specified code of the hypervisor and the monitor is therefore still contained in their allocated blocks. This is the third and last property of *SOUND_MMU*. Hence, *SOUND_MMU*(*i'*) holds.
- *LINUX*(*i'*): All but the first conjunct of *LINUX* depend only on τ and the contents of *L1* and *L2* blocks. Neither of these state components can be modified by non-privileged CPU transitions, since only specification transitions can modify τ, and according to *CONST_PT*, non-privileged CPU transitions cannot modify *L1* nor *L2* blocks.

  The first conjunct of *LINUX*, "*If the program counter is mapped by the MMU to a physical address allocated to Linux, then the CPU is in non-privileged mode with DACR[5:4] = 0b01.*", is the informal description of *LCE*(*i*) ⇒ *CPUL*(*i*). The following reasoning of why this predicate is preserved is done by case analysis of when *LCE*(*i*) is true and false.

  Assume *LCE*(*i*) is true. Since *LCE*(*i*) ⇒ *CPUL*(*i*) holds, *CPUL*(*i*) holds. *CPUL*(*i*) depends only on *CPSR*[4:0] and *DACR*[5:4]. Of these two register fields only *CPSR*[4:0] can be modified when the CPU is in non-privileged mode. Such modifications can only occur due to the CPU taking an exception. Exceptions cause the program counter to be set to a specific address. Since *SOUND_MMU*(*i'*) states that the hypervisor code is mapped as specified. This means that an exception causes the MMU to map the program counter to hypervisor code. Hence, when exceptions occur *LCE*(*i'*) is false, and when no exceptions occur *CPUL*(*i'*) is true since *CPUL*(*i*) is true and *CPSR*[4:0] and *DACR*[5:4] are unmodified. Irrespective of whether an exception occur, *LCE*(*i'*) ⇒ *CPUL*(*i'*) is true.

Assume that *LCE*(*i*) is false. By the definitions of *LCE* and *mmu*, when *LCE*(*i*) is false *mmu*(*i*, …) either returns ⊥ or a physical address that is outside the memory region allocated to Linux. Consider first the case when *mmu*(*i*, …) returns ⊥. By the definition of *mmu*, this means that either the virtual address in the program counter is unmapped or mapped without execute access permission. In either case, the CPU takes an exception. In the previous paragraph it was reasoned that when an exception occur, *LCE*(*i'*) is false. Hence, if *mmu*(*i*, …) returns ⊥, *LCE*(*i'*) ⇒ *CPUL*(*i'*) is true.

Consider the case when *mmu*(*i*, …) returns a physical address that is outside the memory region allocated to Linux. By *LINUX*(*i*) blocks of type *L1*, *L2* or *D* are allocated to Linux. The physical address returned by *mmu*(*i*, …) must therefore be mapped to a physical address that belongs to a block of type ⊥, *MN* or *N*. Such blocks are mapped as non-executable as implied by the following predicates:

- *SOUND_MMU*(*i*) and *SOUND_PT*(*i*): As was reasoned for *SOUND_PT*(*i'*), these two predicates imply that *mmu*(*i*, …) depends only on page tables in *L1* and *L2* blocks.
- *LINUX*(*i*): Page table entries in L1 and L2 blocks maps blocks typed as ⊥ as inaccessible to the CPU when the CPU executes Linux (in the ideal model the CPU executes Linux when the CPU is in non-privileged mode).
- *LINUX*(*i*): NIC registers are located in blocks typed as *MN* or *N*, and which are the only blocks typed as *MN* or *N*.
- *NIC_READ_ONLY*(*i*): When the CPU is configured to execute Linux (which the CPU is when it is in non-privileged mode in the ideal model), the CPU cannot interpret the contents of NIC registers as instructions to execute.

Hence, blocks of type ⊥, *MN* or *N*, are mapped as non-executable causing *mmu*(*i*, …) to return ⊥. This leads to a contradiction since ⊥ is not a physical address, which was assumed.

The implication *LCE*(*i'*) ⇒ *CPUL*(*i'*) is therefore true irrespective of whether *LCE*(*i*) is true or false, meaning that *LINUX*(*i'*) holds.

The following motivates why non-privileged CPU transitions preserve all predicates of *NIC*. By the definition of the ideal model only specification transitions can access the *i.spec* state component. Non-privileged CPU transitions therefore cannot modify the *i.spec* state component.

NIC_READ_ONLY implies that non-privileged CPU transitions cannot write NIC registers affecting which memory accesses the NIC performs. Since the NIC model only includes NIC registers affecting which memory accesses the NIC performs, this means that non-privileged CPU transitions cannot modify the *i.nic* state component by writing NIC registers.

NIC_READ_ONLY also implies that non-privileged CPU transitions cannot read the physical address space at unaligned physical addresses locating parts of NIC registers. Such reads are the only reads that can modify the *i.nic* state component (which would cause the NIC model to modify the *i.nic* state component by setting *i.nic.dead* to true, marking the NIC state as dead and undefined). This means that non-privileged CPU transitions cannot modify the *i.nic* state component by reading NIC registers.

Furthermore, *TTBR0*, and *DACR* cannot be modified by non-privileged CPU transitions. As was reasoned for *SOUND_PT*(*i'*), non-privileged CPU transitions cannot modify the contents of L1 and L2 blocks.

Non-privileged CPU transitions therefore cannot modify *TTBR0*, *DACR*, contents of *L1* and *L2* blocks, *nic* nor *spec*. Since *NIC* only depends on *TTBR0*, *DACR*, contents of *L1* and *L2* blocks, *nic* and *spec*, according to the list in Section E.4, non-privileged CPU transitions preserve *NIC*, and therefore *NIC*(*i'*) holds.

6.3.3.2 NIC Transitions Preserve SEC

This subsection motivates why NIC transitions, $i \rightarrow_{NIC} i' \in IM.\delta$, preserve *SOUND_PT*, *CONST_PT*, *SOUND_MMU* and *LINUX* of *CPU_MEMORY*, and *NIC_READ_ONLY* and *CANNOT_DIE* of *NIC*. Some of the reasonings in this subsection depend on the *Constant Memory Lemma* (which is a sub-level lemma motivated in Section F.2):

Constant Memory Lemma.

If an execution trace starts from an ideal state satisfying *SEC* and only consists of NIC transitions, then no transition in that trace modifies the contents of an *L1*, *L2* or executable *D* block, nor hypervisor or monitor memory.

The following list motivates why NIC transitions preserve the four predicates of *CPU_MEMORY*:

- *SOUND_PT* and *CONST_PT*: These two predicates depend only on the contents of *L1*, *L2* and executable *D* blocks and the *i.spec* state component (according to the list in Section E.4). According to the *Constant Memory Lemma*, these blocks are not modifiable by NIC transitions, and NIC transitions cannot modify the *i.spec* state component.
- *SOUND_MMU*: According to the list in Section E.4, this predicate depends only on τ, *TTBR0*, *DACR*, and the contents of *L1*, *L2*, hypervisor and monitor code blocks. By the definition of the ideal model, NIC transitions cannot modify CPU registers nor τ in the *i.spec* state component, and by the

Constant Memory Lemma NIC transitions cannot modify *L1*, *L2*, hypervisor nor monitor code blocks.

- *LINUX*: All but the first conjunct of *LINUX* depend only on τ and the contents of *L1* and *L2* blocks. By the *Constant Memory Lemma*, and since NIC transitions cannot modify τ, it can be concluded that NIC transitions preserve all but the first conjunct of *LINUX*. For the first conjunct, formally described as $LCE(i) \Rightarrow CPUL(i)$, *LCE* depends only on *mmu* and *r15*, and *CPUL* depends only on *CPSR*[4:0] and *DACR*[5:4], where *mmu* depends only on *TTBR0*, *DACR*, contents of *L1* and *L2* blocks, and in order to determine which blocks are typed as *L1* and *L2*, also τ. CPU registers and τ can only be modified by non-privileged CPU or specification transitions, and by the *Constant Memory Lemma*, NIC transitions cannot modify the contents of *L1* or *L2* blocks. Hence, NIC transitions cannot modify any of the state components that the first conjunct of *LINUX* depends on, and it can be concluded that NIC transitions cannot modify the state components that *LINUX* depends on. *LINUX* is therefore preserved by NIC transitions.

The following list motivates why NIC transitions preserve the two predicates of *NIC*:

- *NIC_READ_ONLY*: According to Section D.4, depends only on *TTBR0*, *DACR*, contents of *L1* and *L2* blocks, and *spec*. NIC transitions cannot modify CPU registers nor *spec*, and by the *Constant Memory Lemma*, nor contents of *L1* and *L2* blocks.
- *CANNOT_DIE*: States that autonomous NIC transitions cannot cause the NIC model to enter an undefined state. If the device model framework is correct, the NIC model also cannot enter an undefined state due to a memory read request reply transition. (If the device model framework gives the NIC model a memory read request reply that, for instance, corresponds to a physical address that the NIC model has not requested to access, the NIC model enters an undefined state). Assuming that the device model framework is correct, this means that the NIC model still cannot enter an undefined state after the NIC model has generated an autonomous transition and possibly a following memory read request reply transition.

6.3.3.3 Lemma III Implied by Lemma I and Lemma II

The reasonings in the previous two Subsections 6.3.3.1 and 6.3.3.2 give an indication of why certain predicates of *SEC* are preserved by non-privileged CPU and NIC transitions. Since the initial states of the ideal model, *IM.IS*, are defined to satisfy *SEC* and all transitions in the ideal model, *IM.δ*, preserve *SEC* (which has been reasoned to a significant extent), Lemma II follows:

$$\forall \mathrm{i} \in IM.S.\ SEC(i).$$

Lemma I and Lemma II imply Lemma III

$$\forall i \in IM.S.\ EXEC\_SIGNED\_LINUX\_CODE(i) \land \neg i.nic.dead.$$

This lemma states that in the ideal model, only signed Linux code can be executed and the NIC is always in a defined state. This means that the formal software design is correct and that the formal software design can be considered as the formal specification for the implementation of the hypervisor and the monitor. The next section describes the second part of the proof plan of how it can be proved that the implementation of the hypervisor and the monitor is correct with respect to their formal specification (software design).

6.4 Lemma VI: Implementation Is Correct

The purpose of this section is to describe the details of the second part of the proof plan and how Lemma VI can be proved:

$$\forall r \in RLM.S.\ \exists i \in ILM.S.\ r\ R\ i.$$

This lemma states: For each state in the real model in which the CPU is configured to execute Linux, there exists a state in the ideal model in which the CPU is configured to Linux such that the two states are related by *R*. Two states related by *R* means that those two states have equal values of the state components that affect (i) the sets of transitions starting from the two states, and (ii) the validity of the property of Lemma III for an individual ideal state. The first property is used to prove that the executions of the binary code of Linux are identical on the binary interfaces described by the real model and the ideal model. The second property is used to transfer the property of Lemma III from a state in the ideal model to a state in the real model.

The presentation of this section follows the four steps outlined in Subsection 6.2.2.2. Subsection 6.4.1 presents the definition of the simulation relation *R* such that *R* states the equality described in the previous paragraph. Subsection 6.4.2 defines *RLM* and *ILM* on top of *RM* and *IM*, respectively, to only include states and transitions where the CPU is configured to execute Linux and transitions describing the complete execution of an exception handler. Subsection 6.4.3 states Lemma IV and motivates that lemma by means of a sub-level lemma which describes how it can be proved that each initial state in *RLM.IS* is related to some initial state in *ILM.IS*. Subsection 6.4.4 describes how the simulation proof method is applied to prove Lemma V of that each transition is *RLM.δ* is matched by a transition in *ILM.δ* with respect to *R*. Subsection 6.4.5 concludes by describing how Lemma VI is implied by Lemma IV and Lemma V.

Some execution trace notation is useful in this section. Let

$$\pi = s_0 \rightarrow_{l\_0} s_1 \rightarrow_{l\_1} \dots \rightarrow_{l\_n\text{-}1} s_n$$

be a sequence of transitions in *RM.δ*, *IM.δ*, *RLM.δ* or *ILM.δ*, respectively. Then:

- $\pi[a{:}b]$, $0 \leq a \leq b \leq n$, is the sub-execution trace of π starting from the a-th state s_a and ending in the b-th state s_b: $\pi[a{:}b] = s_a \rightarrow_{l\_a} \dots \rightarrow_{l\_b\text{-}1} s_b$.
- $\pi_{[a]}$, $0 \leq a \leq n$, is the a-th state of π: $\pi_{[a]} = s_a$.
- $label(\pi, a)$, $0 \leq a < n$, is the label of the a-th transition: $label(\pi, a) = l\_a$.
- $length(\pi)$ is equal to the number of transitions in π: $length(\pi) = n$.

6.4.1 Definition of Simulation Relation R

For R to be used to transfer Lemma III from the ideal model to the real model, R must require a related pair of a real state and an ideal states to have equal values of certain state components such that:

- Lemma V can be proved: For each transition in $RLM.\delta$ from a real state in $RLM.S$, there exists a transition in $ILM.\delta$ from a related ideal state in $ILM.S$ such that the two transitions describe identical operations.
- Lemma IX can be proved: The properties of Lemma III, if Linux code is executed then that code is signed, and the NIC is in a defined state, can be transferred by R from the ideal state to the related real state.

During the development of the proof plan these two requirements of R have lead to the following definition of $R \subseteq S_{real} \times S_{ideal}$:

$$R \overset{\text{def}}{=} \{(r, i) \mid CPU\_EQ(r.cpu, i.cpu) \wedge MEMORY\_EQ(r.memory, i.memory) \wedge \\ NIC\_EQ(r.nic, i.nic) \wedge HVM\_SPEC\_EQ(r.memory, i.spec)\},$$

where:

- bool CPU_EQ(cpu_state $rcpu$, cpu_state $icpu$) $\overset{\text{def}}{=}$
  $$rcpu.uregs = icpu.uregs \wedge rcpu.sregs.CPSR = icpu.sregs.CPSR \wedge \\ rcpu.cp15.TTBR0 = icpu.cp15.TTBR0 \wedge \\ rcpu.cp15.DACR = icpu.cp15.DACR.$$
- bool $MEMORY\_EQ$(word32 → word8 $rmemory$, word32 → word8 $imemory$) $\overset{\text{def}}{=}$
  $$[\forall pa \in LINUX\_MEM.\ rmemory(pa) = imemory(pa)] \wedge \\ [\forall pa \in HYP\_CODE.\ rmemory(pa) = imemory(pa)] \wedge \\ [\forall pa \in MON\_CODE.\ rmemory(pa) = imemory(pa)],$$

  where HYP_CODE and MON_CODE are the sets of physical addresses allocated to the hypervisor and the monitor to store their code, respectively. The description of the predicate $SOUND\_MMU$ in Subsection 6.3.1 mentions that the contents of these two memory regions contain the code of the hypervisor and the monitor as produced by the compilation of their source code. This means that the ideal model specifies (and contains) the binary code implementing the hypervisor and the monitor according to what the compiler produces from their source code.
- bool NIC_EQ(nic_state $rnic$, nic_state $inic$) $\overset{\text{def}}{=} rnic = inic$.
- bool HVM_SPEC_EQ(word32 → word8 $memory$, spec_state $spec$) $\overset{\text{def}}{=}$
  $$hvm\_to\_spec(memory) = spec.$$

The rest of this subsection gives an intuition of why this definition of R can be used to prove Lemma V and Lemma IX. Since Lemma V assumes that the related ideal state is in the ideal model, $IM.S$, the ideal state satisfies SEC by Lemma II. The following properties, implied by SEC, CPU_EQ, $MEMORY\_EQ$ and NIC_EQ therefore hold on the related states:

- SEC implies that the page tables used by the MMU in the ideal state are located in the memory region allocated to Linux and that those page tables only allow the CPU in non-privileged mode to access Linux memory and read NIC registers: *SOUND_MMU* states that the first-level page table used by the MMU is located in an *L1* block. *SOUND_PT* states that second-level entries of page tables located in *L1* blocks only refer to page tables in *L2* blocks. *LINUX* states that all blocks typed as *L1*, *L2* or *D* are allocated to Linux and blocks typed as ⊥ are inaccessible. *NIC_READ_ONLY* implies that the CPU when executing Linux (which is done in non-privileged mode) can only access NIC registers by means of reads.

- *CPU_EQ* implies that the location of the first-level page table used by the MMU, and the values of the CPU registers that affect the execution of CPU instructions in non-privileged mode, in the related states are equal: *TTBR0* determines the location the first-level page table, and the registers affecting CPU instruction executions in non-privileged mode are the general-purpose registers, *CPSR*, *TTBR0* and *DACR* (there are probably many other registers that affect the execution of CPU instructions in non-privileged mode, but those registers are not critical in this context and therefore can be simply included in the relation).
- *MEMORY_EQ* states that the contents of the memory region allocated to Linux in the related states are equal.
- *NIC_EQ* states that the NIC is in the same state in the related states.

The consequence of these properties is that in related states, the MMU uses identical page tables, which allow the CPU when executing Linux to only access Linux memory and reading NIC registers. Since CPU registers, Linux memory and NIC registers have equal contents in related states, the CPU in the real model and the CPU in the ideal model execute identical Linux code and those executions operate on identical Linux data and NIC register contents. Non-privileged CPU transitions from related states therefore describe identical operations. Hence, Lemma V can be proved for transitions in *RLM.δ* that correspond to transitions in *RM.δ* that start from states in which the CPU is in non-privileged mode.

According to *NIC_EQ*, the nic state components of the related states are equal. Since the transitions of the NIC model only depend on the state of the NIC, this predicate implies that NIC transitions starting from related states describe identical operations. Lemma V can therefore be proved for NIC transitions in *RLM.δ*.

Consider the properties of the following assumption and the definitions of *MEMORY_EQ* and *HVM_SPEC_EQ*:

- Assumption: The executions of the binary code of the hypervisor and the monitor contained in their memory regions in the memory state component of the ideal state, and the specification transitions describe identical operations.
- *MEMORY_EQ*: The binary code of the hypervisor and the monitor in the related states are equal.
- *HVM_SPEC_EQ*: The values of the data structures of the hypervisor and the monitor in the related states are equal.

These three properties imply that the executions of the exception handlers of the hypervisor and the monitor in the real model and the specification transitions in the ideal model, describe identical operations. Lemma V can therefore be proved for transitions in *RLM.δ* that correspond to exception handler executions. Hence, Lemma V can be proved for all three types of transitions in *RLM.δ*: CPU transitions describing the execution of Linux, NIC transitions describing the execution of the NIC, and exception handler transitions describing the execution of exception handlers possibly interleaved with NIC transitions.

Consider Lemma IX. Since Lemma IX assumes that the related ideal state is in *IM.S*, the ideal state satisfies *SEC* by Lemma II. Lemma I can therefore be applied stating that from the ideal state, if the CPU executes Linux code, that code is signed. Since the ideal state satisfies *SEC*, the properties implied by *SEC*, *CPU_EQ*, *MEMORY_EQ* and *NIC_EQ* described above hold on the related states, and therefore the CPU executes identical Linux code from the related states. Hence, only signed Linux code is executed from the related real state.

By *NIC_EQ* the NIC is in the same state in the related states. Since the NIC is in a defined state in the ideal state, according to Lemma I, the NIC is also in a defined state in the real state.

6.4.2 Definitions of RLM and ILM

Proving Lemma VI with respect to *RM* and *IM* is not possible because the execution of an exception handler of the hypervisor and the monitor is described by several transitions in *RM.δ*, while the execution of an exception handler in the ideal model is described by one specification transition in *IM.δ*. The implication of this is that intermediate states in execution traces describing exception handler executions in *RM* are not related to any state in *IM.S*. For instance, in *RM*, the data structures of the software design are updated stepwise by several transitions, while in *IM* the data structures of the formal specification are updated atomically by one specification transition. In some states in *RM.S* the data structures therefore have inconsistent values, which is never the case for the states in *IM.S*. For instance, in Section 3.4 it is described that the data structures *rx0_active_queue* and *α* has the following relationship: *α*(*w*) is true if and only if the 32-bit word in CPPI_RAM with index *w* is occupied by a buffer descriptor located in the buffer descriptor queue starting at the physical address stored in *rx0_active_queue*. Since *rx0_active_queue* and *α* cannot be updated simultaneously by one CPU transition, this relationship will not hold immediately after one of them has been updated. In *IM*, *rx0_active_queue* and *α* are updated simultaneously by one specification transition. *HVM_SPEC_EQ* is therefore false for certain pairs of states in RM.S and *IM.S*, where the state in *RM.S* is an intermediate state in the execution of an exception handler in *RM.Π*. Similar problems occur for *CPU_EQ*, *MEMORY_EQ*, and *NIC_EQ*, when exception handlers in *RM* modify CPU registers, update page table entries in Linux memory, or write NIC registers.

To address this issue Lemma IV, V and VI are instead proved with respect to the labeled transition systems the real Linux model and the ideal Linux model. These two labeled transition systems are referred to by the four tuples *RLM* and *ILM* (see also Subsection 6.2.2.2). *RLM* and *ILM* are defined in terms of *RM* and *IM*, respectively, but describe the executions of exception handlers by one transition. These definitions of *RLM.S* and *ILM.S* exclude intermediate states of exception handler executions, thereby making it possible to prove Lemma VI: For each real state r in *RLM.S* there exists an ideal state i in *ILM.S* such that $r\ R\ i$ holds. *RLM.S* and *ILM.S* are therefore defined as *RM.S* and *IM.S*, except that states in *RM.S* and *IM.S* are omitted for which the CPU is not configured to execute Linux (that is, states not satisfying *CPUL* and which are part of exception handler executions), respectively. *RLM.δ* and *ILM.δ* are then defined to only contain transitions from one such state to another such state, both satisfying *CPUL*, on one condition. The condition is that there exists a sub-execution trace of an execution trace in *RM.Π* or *IM.Π*, respectively, starting from the former state and ending in the other, such that no intermediate state in that sub-execution trace satisfies *CPUL*, except for states immediately following a sequence of transitions describing an exception handler execution and from which NIC transitions start.

For instance, in Figure 40 in Subsection 6.2.2.2, $r_a \rightsquigarrow_{real} r_c$, $r_a \rightsquigarrow_{real} r_{c+1}$, $i_{a'} \rightsquigarrow_{ideal} i_{c'}$ and $i_{a'} \rightsquigarrow_{ideal} i_{c'+1}$ exist in *RLM.δ* or *ILM.δ* because there exist four sub-execution traces of four execution traces in *RM.Π* or *IM.Π* such that in those four sub-execution traces, r_a, r_r, r_{c+1}, $i_{a'}$, $i_{c'}$ and $i_{c'+1}$ satisfy *CPUL* but not the states between them, except for r_c and $i_{c'}$ from which NIC transitions start. Similarly, $r_{a-1} \rightsquigarrow_{real} r_a$, $r_{c+1} \rightsquigarrow_{real} r_{c+2}$, $i_{a'-1} \rightsquigarrow_{ideal} i_{a'}$ and $i_{c'+1} \rightsquigarrow_{ideal} i_{c'+2}$ exist in *RLM.δ* and *ILM.δ* since there exist sub-execution traces of execution traces in *RM.Π* or *IM.Π*, each consisting of a single transition starting from and ending in the involved states, which satisfy *CPUL*. $r_c \rightsquigarrow_{real} r_{c+1}$ and $i_{c'} \rightsquigarrow_{ideal} i_{c'+1}$ are not in *RLM.δ* and *ILM.δ* since they correspond to NIC transitions in *RM.δ* and *IM.δ*, respectively, immediately following sequences of transitions describing exception handler executions.

Notice that *RLM.S* and *ILM.S* only include states from which Linux code *can* be executed, since all states in *RLM.S* and *ILM.S* satisfy *CPUL*. To prove that only signed Linux code is executed, it is sufficient to only consider such states. However, Theorem I is not defined in terms of *RLM* since *RLM* does not describe the real model implemented in HOL4. Also, *RLM* is less accurate than *RM* since *RLM* does not describe (include) certain critical hardware system executions. For instance, *RLM* does not describe executions of exception handlers that sets the program counter to Linux memory when the CPU is in non-privileged mode. Such exception handler executions give the control of the system to Linux, most likely causing *CPUL* to never hold again. Since *RLM.Π* only contains execution traces consisting of transitions between states satisfying *CPUL*, such behavior of exception handler executions are omitted from *RLM.δ* and *RLM.Π*. This execution behavior of exception handlers must be included in the model and must be proved to not occur. This issue is treated by the proof of Lemma VII in Subsection 6.5.1.

RLM is the four tuple $RLM \overset{\text{def}}{=} (S, IS, \delta, \Pi)$ where each component is defined as follows:

- $RLM.S \overset{\text{def}}{=} \{r \mid r \in RM.S \land CPUL(r)\}$: The set of states in the real Linux model is the set of states in the real model that satisfy *CPUL*.
- $RLM.IS \overset{\text{def}}{=} RM.IS$: The set of initial states in the real Linux model is equal to the set of initial states in the real model. This definition of *RLM.IS* is consistent with the definition of *RLM.S*, motivated as follows. The sub-

level lemma *RM and IM Initially Related Lemma* states (as motivated in Section F.3):

$$\forall r \in RM.IS.\ \exists i \in IM.IS.\ r\ R\ i.$$

Consider an arbitrary real state r in *RM.IS*. By the definition of *IM.IS* (which is equal to IS_{ideal}, see Subsection 5.3.3), $SEC(i) \land LCE(i)$ holds. Since $SEC(i)$ implies $LINUX(i)$, which in turn implies $LCE(i) \Rightarrow CPUL(i)$ (see the first bullet item in the description of *LINUX* in Subsection 6.3.1), $CPUL(i)$ holds. R requires r and i to have equal values of the *CPSR* and *DACR* registers. Since *CPUL* only depends on these registers, $CPUL(r)$ holds. This means that r both is in *RM.S* and satisfies *CPUL*, implying r is in *RLM.S*. That is, *RM*.IS ⊆ *RLM.S* and therefore *RLM.IS* ⊆ *RM.IS*.

- $RLM.\delta \subseteq S_{real} \times \{real\} \times S_{real}$: The set of transitions in the real Linux model. Each transition in $RLM.\delta$ has the label *real*. $(r, real, r') \in RLM.\delta$ is written as $r \rightsquigarrow_{real} r'$. $r \rightsquigarrow_{real} r'$ holds if and only if the following predicate holds:

  $\exists \pi \in RM.\Pi,\ 0 \leq a < e < length(\pi).$
  $\qquad r = \pi_{[a]} \land r' = \pi_{[e]} \land CPUL(\pi_{[a]}) \land CPUL(\pi_{[e]}) \land$
  $\qquad [LT(\pi, a, e) \lor EHT(\pi, a, e) \lor NT(\pi, a, e)],$

  where *LT*, *NT* and *EHT*, together with the first four conjunctions, states that $r \rightsquigarrow_{real} r'$ corresponds to either a Linux transition, an exception handler execution, or a NIC transition, respectively. That is, $r \rightsquigarrow_{real} r'$ corresponds to a sub-execution trace $\pi[a{:}e]$ in $RM.\Pi$ that is of one of the following three types:

  - A CPU transition describing the execution of a CPU instruction located in Linux memory.
  - A sub-execution trace consisting of (i) transitions describing an execution of an exception handler of the hypervisor and the monitor, and (ii) the longest existing consecutive sequence of NIC transitions following that exception handler execution sub-trace. The reason why the transition $r \rightsquigarrow_{real} r'$ not only corresponds to the execution of an exception handler but also includes trailing NIC transitions is motivated in the description of the definition of $ILM.\delta$.

    However, for each transition $r \rightsquigarrow_{real} r'$ corresponding to an exception handler execution and a non-empty sequence of NIC transitions in an execution trace π, there always exists another execution trace π' giving rise to the existence of a transition $r \rightsquigarrow_{real} r''$ where r'' is followed by a NIC transition in π but a CPU transitions in π'. The reason it that the non-deterministic scheduler of the device model framework can select either a NIC transition or a CPU transition from r''. In π, a NIC transition was selected but in π', a CPU transition was selected. Hence, a definition where an arbitrary number of NIC transitions follows the exception handler transition would be equivalent with respect to which transitions are defined to be in $ILM.\delta$. This definition of exception handler execution transitions was chosen because this definition specifies which transitions are wanted in $ILM.\delta$ with respect to a specific execution trace π.

- A NIC transition describing one or two execution steps of the NIC (autonomous NIC transition possibly followed by a memory read request reply transition) between two states in which the CPU is configured to execute Linux. In addition, that NIC transition is not part of a consecutive sequence of NIC transitions immediately following a sub-execution trace describing an execution of an exception handler. Such NIC transitions are omitted in order to not include superfluous transitions in $RLM.\delta$ not needed in the proof plan (see previous bullet item; this is not important but gives a more precise set of transitions).

Figure 43 gives a graphical illustration of which sub-execution traces $\pi[a{:}e]$, a transition $r \rightsquigarrow_{real} r'$ might correspond to. In the topmost sub-execution trace, $r \rightsquigarrow_{real} r'$ corresponds to one CPU transition describing an execution of a CPU instruction located in Linux memory, since the start state and the end state satisfy $CPUL$. In the second topmost sub-execution trace, $r \rightsquigarrow_{real} r'$ corresponds to one NIC transition, since the start state and the end state satisfy $CPUL$, and the NIC transition is not a part of a consecutive sequence of NIC transitions immediately following a sub-execution trace describing an execution of an exception handler. In the second bottom sub-execution trace, $r \rightsquigarrow_{real} r'$ corresponds to a sub-execution trace describing an exception handler execution followed by the longest existing consecutive sequence of NIC transitions in π following $\pi_{[c]}$. The bottom sub-execution trace is similar to the second bottom sub-execution trace but with the difference that the longest existing consecutive sequence of NIC transitions is empty. In this case, $c = e$.

Recall that the non-deterministic scheduler of the device model framework always can schedule the CPU model to describe the next transition from a certain state. This means that for each sub-execution trace $\pi[a{:}e]$ defining an exception handler transition including trailing NIC transitions in $\pi[c{:}e]$, there always exists a sub-execution trace $\pi'[a{:}e]$ of another execution trace π' such that, $\pi[0{:}d] = \pi'[0{:}d]$, for some $c \leq d < e$, $label(\pi, d) = NIC$ and $label(\pi', d) = CPU$. Hence, for each state followed by a NIC transition and being a part of a sequence of NIC transitions immediately following a transition sequence of an exception handler execution, with respect to a certain sub-execution trace, there always exists a transition in $RLM.\delta$ that ends in that state. In Figure 40 in Subsection 6.2.2.2, r_c is such a state which is the last state of a transition sequence of an exception handler execution and which is followed by a single NIC transition, which in turn is followed by a CPU transition, defining the transition $r_a \rightsquigarrow_{real} r_{c+1}$. Since there exists another execution trace in which r_c is followed by a CPU (or exception) transition, the transition $r_a \rightsquigarrow_{real} r_c$ also exists in $RLM.\delta$, ending in r_c.

The following list formally defines and explains LT, EHT and NT, where EHT is defined in terms of EH and TNT:

- bool LT(real_trace π, nat a, nat e) $\overset{\text{def}}{=} a + 1 = e \wedge label(\pi, a) = CPU$.

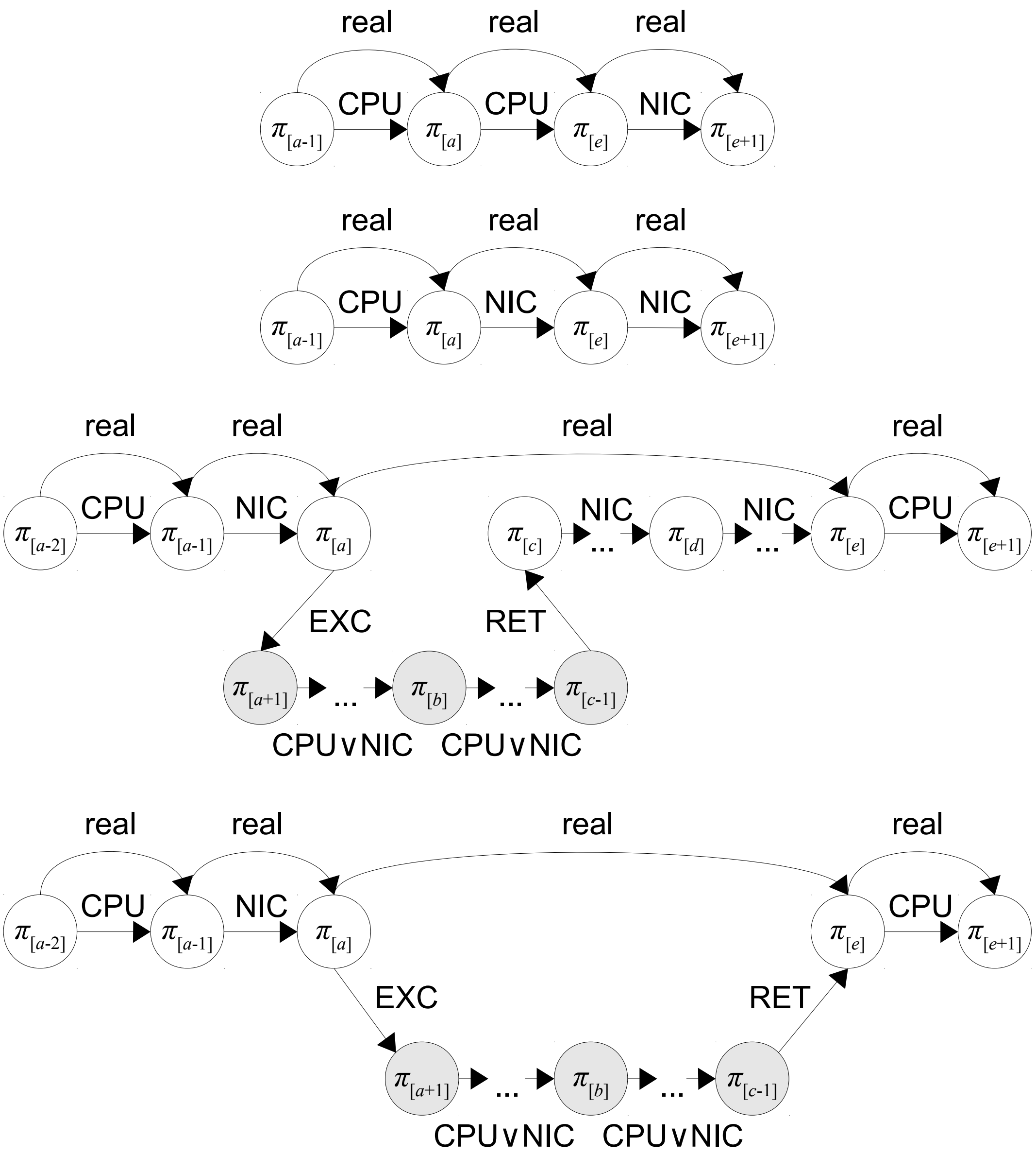


Figure 43: A graphical illustration of which sub-execution traces $\pi[a:e]$, $\pi \in RM.\Pi$, a transition $r \rightsquigarrow_{real} r' \in RLM.\delta$ can correspond to. The white states satisfy CPUL which the shaded states do not. Four sub-execution traces are shown, denoted by $\pi[a\text{-}1:e+1]$ or $\pi[a\text{-}2:e+1]$. For each sub-execution trace, $\pi[a:e]$ corresponds to a transition $r \rightsquigarrow_{real} r' \in RLM.\delta$.

LT (Linux Transition) states that $\pi[a{:}e]$ is one CPU transition. (real_trace is the data type for sequences of transitions between real states of the data type real_state, and nat is the data type for natural numbers, including zero.)

- bool EH(real_trace π, nat a, nat c) $\overset{\text{def}}{=}$
  $[\exists c.\ a < b < c] \land [\forall a < b < c.\ \neg CPUL(\pi_{[b]})] \land CPUL(\pi_{[c]})$.

    EH (Exception Handler) states that:

    - There exists at least one state in π between $\pi_{[a]}$ and $\pi_{[c]}$.
    - All states in π between $\pi_{[a]}$ and $\pi_{[c]}$ does not satisfy *CPUL*, and $\pi_{[c]}$ satisfies *CPUL*, meaning that $\pi[a + 1{:}c - 1]$ is a part of an exception handler execution trace, and $\pi_{[c]}$ is the state to which that exception handler execution returns and in which the CPU is configured to execute Linux.

  - bool $TNT(\text{real\_trace } \pi, \text{nat } c, \text{nat } e) \overset{\text{def}}{=}$
    $[\forall c \leq d < e.\ label(\pi, d) = NIC] \wedge label(\pi, e) \neq NIC.$

    TNT (Trailing NIC Transitions) states that all transitions in $\pi[c{:}e]$, if any ($c < e$), are NIC transitions, but not the following transition $\pi[e{:}\ e + 1]$.

  - bool: $EHT(\text{real\_trace } \pi, \text{nat } a, \text{nat } e) \overset{\text{def}}{=}$
    $\exists a < c \leq e.\ EH(\pi, a, c) \wedge TNT(\pi, c, e).$

    EHT (Exception Handler sub-execution Trace) states that $\pi[a{:}e]$ is an execution trace of an exception handler execution, $\pi[a{:}c]$, followed by the longest consecutive sequence of NIC transitions, $\pi[c{:}e]$, which is possibly empty.

  - bool $NT(\text{real\_trace } \pi, \text{nat } a, \text{nat } e) \overset{\text{def}}{=}$
    $a + 1 = e \wedge label(\pi, a) = NIC \wedge$
    $\neg[\exists 0 \leq f < a.\ label(\pi, f) = RET \wedge [\forall f < g \leq a.\ label(\pi, g) = NIC]].$

    NT (NIC Transition) states that $\pi[a{:}e]$ is one NIC transition, and that this transition is not a part of a sub-execution trace of π only consisting of NIC transitions immediately following a sub-execution trace describing an exception handler execution.

- *RLM.Π*: The set of execution traces in the real Linux model. Let

  $$\psi = r_0 \rightsquigarrow_{real} \ldots \rightsquigarrow_{real} r_n,\ n \geq 0.$$

  ψ is in *RLM.Π* if and only if $r_0 \in RLM.IS$ and each transition is in $RLM.\delta$: $\forall 0 \leq j < n.\ r_j \rightsquigarrow_{real} r_{j+1} \in RLM.\delta$.

The ideal Linux model, $ILM \overset{\text{def}}{=} (S, IS, \delta, \Pi)$, is defined similarly:

- $ILM.S \overset{\text{def}}{=} \{i \mid i \in IM.S \wedge CPUL(i)\}$.
- $ILM.IS \overset{\text{def}}{=} IM.IS$. This definition is consistent with the definition of *ILM.S*. The states in *IM.IS* are defined to satisfy *LCE* and *SEC*, where *SEC* includes the implication $LCE \Rightarrow CPUL$. All states in *IM.IS* therefore satisfy *CPUL*. Hence, $ILM.IS \subseteq ILM.S$.
- $ILM.\delta \subseteq S_{ideal} \times \{ideal\} \times S_{ideal}$: All transitions in the ideal Linux model have the label *ideal*. $(i, ideal, i') \in ILM.\delta$ is written as $i \rightsquigarrow_{ideal} i'$. $ILM.\delta$ is defined as $RLM.\delta$ is defined, but where the involved predicates, *LT*, *EH*, *TNT*, *EHT* and *NT* are defined for execution traces consisting of ideal states. $i \rightsquigarrow_{ideal} i'$ holds if and only if the following predicate holds:

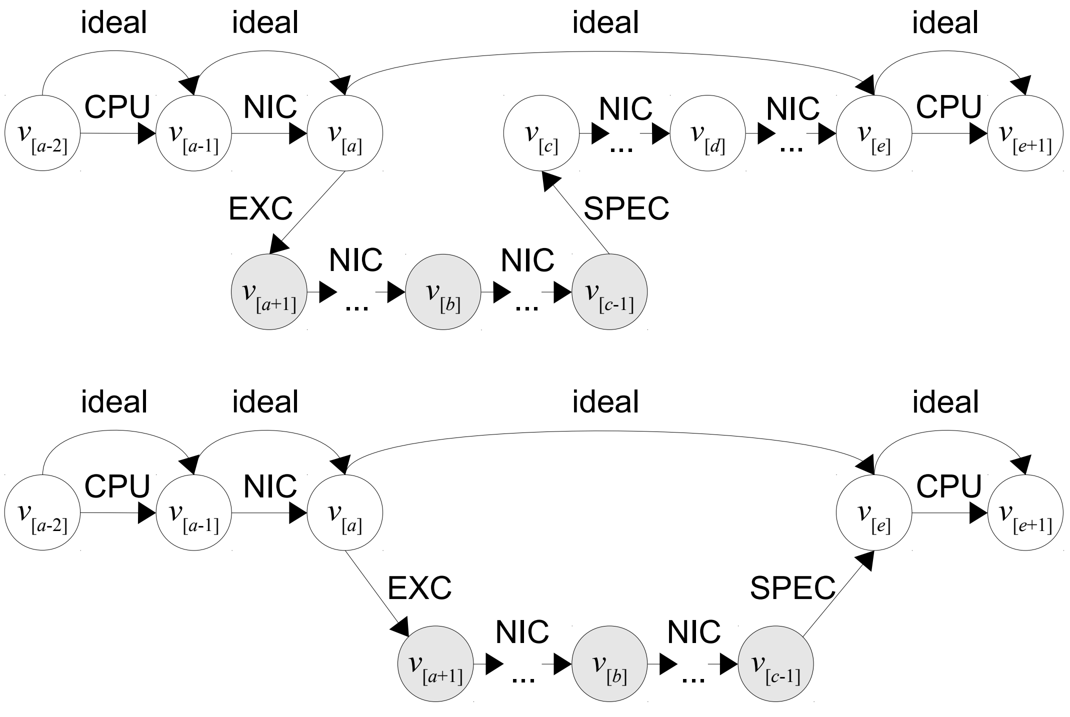


Figure 44: A graphical representation of the structure of the exception handler transitions in ILM.δ. Two sub-execution traces are shown, denoted by v[a-2:e+1], v ∈ IM.Π. For both sub-execution traces, v[a:e] defines an exception handler transition i $\rightsquigarrow_{ideal}$ i' ∈ ILM.δ. Comparing to Figure 43, there are two differences between exception handler transitions in RLM.δ and exception handler transitions in ILM.δ. The first difference is that for r $\rightsquigarrow_{real}$ r', π[a+1:c-1] can contain both CPU and NIC transitions, while for i $\rightsquigarrow_{ideal}$ i', v[a+1:c-1] can only contain NIC transitions. The second difference is that for r $\rightsquigarrow_{real}$ r', the transition π[c-1:c], having the label RET, describes an execution of an exception return instruction, while for i $\rightsquigarrow_{ideal}$ i', the transition v[c-1:c], having the label SPEC, describes the execution of all operations of an exception handler. Linux and NIC transitions have identical structure in RLM.δ and ILM.δ, corresponding to one CPU transition or one NIC transition in RM.δ or IM.δ, respectively.

$$\begin{aligned}
&\exists v \in IM.\Pi,\ 0 \leq a < e < length(v).\\
&\qquad i = v_{[a]} \wedge i' = v_{[e]} \wedge CPUL(v_{[a]}) \wedge CPUL(v_{[e]}) \wedge\\
&\qquad [LT(v, a, e) \vee EHT(v, a, e) \vee NT(v, a, e)].
\end{aligned}$$

Figure 44 gives a graphical representation of which sub-execution traces $v[a{:}e]$ can correspond to if $EHT(v, a, e)$ holds, defining an exception handler transition $i \rightsquigarrow_{ideal} i' \in ILM.\delta$.

Since specification transitions in *IM* include descriptions of exception returns, *NT* is defined slightly differently for execution traces in *IM.Π* compared to execution traces in *RM.Π*:

$$\begin{aligned}&\text{bool } NT(\text{ideal\_trace } v, \text{ nat } a, \text{ nat } e) \overset{\text{def}}{=} \\ &\quad a + 1 = e \land label(v\ , a) = NIC] \land \\ &\quad \neg[\exists 0 \leq f < a.\ label(v\ ,f) = SPEC \land [\forall f < g \leq a.\ label(v\ , g) = NIC]].\end{aligned}$$

The reason why transitions in *RLM.δ* and *ILM.δ* corresponding to exception handler sub-execution traces in *RM.Π* and *IM.Π* are defined to also include the longest following consecutive sequence of NIC transitions, is because of the proof approach of Lemma V. As briefly described in Subsection 6.2.2.2, Lemma V can be proved for a transition $r \rightsquigarrow_{real} r'$ corresponding to an exception handler sub-execution trace $\pi[a{:}e]$ by first identifying another sub-execution trace $\pi'[a{:}e]$ in *RM.Π* such that the following three conditions hold. First, the start states are equal, $\pi_{[a]} = \pi'_{[a]}$. Second, all transitions in $\pi[a{:}e]$ combined describe identical operations as described by all transitions in $\pi'[a{:}e]$ combined, but in $\pi'[a{:}e]$ the CPU transitions occur in consecutive sequence. Third, since the transitions in $\pi[a{:}e]$ and $\pi'[a{:}e]$ describe identical operations (the operations are possibly described in different orders), $\pi_{[e]} = \pi'_{[e]}$. (In Figure 41, $\pi_{[a]}$ and $\pi'_{[a]}$ are denoted by r_a, and $\pi_{[e]}$ and $\pi'_{[e]}$ are denoted by r_c, where no consecutive sequence of NIC transitions immediately follows the transitions describing the exception handler execution).

In the description of how Lemma V can be proved (sketched in Subsection 6.2.2.2 and described formally in Subsection 6.4.4), it is motivated that for each exception handler sub-execution trace $\pi[a{:}e]$ there exists such another exception handler sub-execution trace $\pi'[a{:}e]$ in *RM.Π*. Due to the requirements that $\pi[a{:}e]$ and $\pi'[a{:}e]$ describe identical operations and that the CPU transitions occur in consecutive sequence in $\pi'[a{:}e]$, some operations described by NIC transitions during the exception handler execution in $\pi[a{:}e]$ are described by NIC transitions after the exception handler execution in $\pi'[a{:}e]$.

Once π' has been identified, Lemma V is proved by identifying a transition $i \rightsquigarrow_{ideal} i'$ corresponding to an exception handler sub-execution trace $v[a'{:}e']$ in *IM.Π*, such that $v[a'{:}e']$ matches $\pi'[a{:}e]$ with respect to R. $v[a'{:}e']$ matches $\pi'[a{:}e]$ with respect to R if $v[a'{:}e']$ describe identical operations as described by $\pi'[a{:}e]$, which is proved in four steps:

1. The *EXC* transitions in $\pi'[a{:}a + 1]$ and $v[a'{:}a' + 1]$ describe identical operations.
2. The following NIC transitions describe identical operations.
3. The CPU transitions (including the *RET* transition) in π' describe identical operations as described by the specification transition in v. The reasoning in this step relies on that the CPU transitions in π' occur in consecutive sequence.
4. The trailing NIC transitions in π' describe identical operations as described by the trailing NIC transitions in v.

It is the fourth proof step that necessitates the need for the transitions in *ILM.δ* that describe exception handler executions to also include all trailing

NIC transitions. This fourth proof step is only necessary when the exception handler sub-execution trace $\pi[a{:}e]$ interacts with the NIC.

However, just because transitions in *ILM.δ* describing exception handler executions must include all trailing NIC transitions, it does not mean that the transitions in *RLM.δ* that describe exception handler executions must also include all trailing NIC transitions. The reason why *RLM.δ* is defined similarly to how *ILM.δ* is defined is because such a definition of *RLM.δ* makes the transitions in *RLM.δ* and *ILM.δ* have a similar structure, making them easier to deal with in the proof of Lemma V. Such a definition of *RLM.δ* also enables a bisimulation proof between the real Linux model and the ideal Linux model. (A bisimulation proof between *RLM* and *ILM* means that Lemma V is also proved for the case where the roles of *RLM* and *ILM* are swapped). Bisimulation results are motivated in Section F.12 and discussed in Section 8.1.

- *ILM.Π*: The set of execution traces in the ideal Linux model. Let

$$\omega = i_0 \rightsquigarrow_{ideal} \ldots \rightsquigarrow_{ideal} i_n,\ n \geq 0.$$

ω is in *ILM.Π* if and only if $i_0 \in ILM.IS$ and each transition of ω is in *ILM.δ*: $\forall 0 \leq j < n.\ i_j \rightsquigarrow_{real} i_{j+1} \in ILM.\delta$.

6.4.3 Lemma IV: Correct Initialization

Lemma IV ($\forall r \in RLM.IS.\ \exists i \in ILM.IS.\ r\ R\ i$) states that for each initial state in the real Linux model, there exists an initial state in the ideal Linux model such that the two states are related by *R*. The meaning of this statement is that the initialization code of the hypervisor initializes the system into a secure state from which Linux can be executed. Subsection F.3 motivates the sub-level lemma *RM and IM Initially Related Lemma*, which states that for each initial state in the real model, there exists an initial state in the ideal model such that the two states are related by *R* ($\forall r \in RM.IS.\ \exists i \in IM.IS.\ r\ R\ i$). Since *RLM.IS* and *ILM.IS* are defined to be equal to *RM.IS* and *IM.IS*, respectively, Lemma IV follows from *RM and IM Initially Related Lemma*.

6.4.4 Lemma V: ILM Simulates RLM

This subsection describes the most important parts in proving Lemma V:

$$\forall r, r' \in RLM.S, i \in ILM.S.\ r \rightsquigarrow_{real} r' \wedge r\ R\ i \Rightarrow \exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \wedge r'\ R\ i'.$$

Lemma V states that for each transition $r \rightsquigarrow_{real} r'$ in the real Linux model, where r is related by *R* to an arbitrary state i in the ideal Linux model, there exists a transition $i \rightsquigarrow_{ideal} i'$ in the ideal Linux model such that r' and i' are related by *R*. (Stated in terms of the terminology used previously: For each transition in the real Linux model there exists a transition in the ideal Linux model such that latter transition matches the former with respect to *R*). The meaning of Lemma V is that the implementation of the exception handlers of the hypervisor and the monitor is correct. It also means that for each execution of the binary code of Linux on the binary interface described by the real Linux model, there exists an identical execution of the binary code of Linux on the binary interface described by the ideal Linux model.

The description of how Lemma V can be proved follows the following pattern:

1. Consider two arbitrary real states *r* and *r'* in *RLM.S* and one arbitrary ideal state *i* in *ILM.S*.
2. Assume that there exists a transition in *RLM.δ* from *r* to *r'*, $r \rightsquigarrow_{real} r'$, and that *r* and *i* are related by *R*, *r R i*.
3. $r \rightsquigarrow_{real} r'$ is of one of three types of transitions in *RLM.δ* due to its definition in terms of *LT*, *NT* and *EHT*:
   - *LT*: $r \rightsquigarrow_{real} r'$ corresponds to one transition in *RM.δ* that describes the execution of a CPU instruction located in Linux memory:
   - *EHT*: $r \rightsquigarrow_{real} r'$ corresponds to an exception handler sub-execution trace $\pi[a{:}e]$ in *RM.Π*, possibly followed by a sequence of NIC transitions.
   - *NT*: $r \rightsquigarrow_{real} r'$ corresponds to one transition in *RM.δ* that describes one or two execution steps of the NIC (one autonomous NIC transition possibly followed by one memory read request reply transition).

   For each of these three types of transitions that $r \rightsquigarrow_{real} r'$ can be of, it is reasoned that there exists an ideal state *i'* in *ILM.S* and a transition in *ILM.δ* from *i* to *i'*, $i \rightsquigarrow_{ideal} i'$, such that *r' R i'* holds. These reasonings rely on several sub-level lemmas, and before the reasonings are described, the referred sub-level lemmas are stated.

6.4.4.1 Sub-Level Lemma Statements

The following sub-level lemmas are referred to in the next three subsections and are included here for easy reference. All of them are motivated in Appendix F. The sub-level lemmas are:

- MMU Lemma.

  If R relates the real model state r to the ideal model state i, then the MMU computes identical values for identical input values in r and i:

  $$\begin{array}{l} \forall r \in RM.S,\ i \in IM.S. \\ \quad r\ R\ i \\ \quad \Rightarrow \\ \quad \forall pl \in \{PL0,\ PL1\},\ va \in \text{<word32>},\ ar \in \{\text{rd, wt, ex}\}. \\ \qquad mmu(r,\ pl,\ va,\ ap) = mmu(i,\ pl,\ va,\ ap). \end{array}$$

- CPU and NIC Rescheduling Lemma.

  Assume that a transition $\pi_{[a]} \rightsquigarrow_{real} \pi_{[e]}$ corresponds to a sub-execution trace $\pi[a{:}e]$ of an exception handler of the hypervisor and the monitor, and where $\pi_{[a]}\ R\ i$ holds for some $i \in IM.S$ and $\pi[a{:}e]$

contains two consecutive transitions. The first transition of those two consecutive transitions is a CPU transition that is not the first nor the last transition in $\pi[a{:}e]$ and that transition does not access a NIC register. The second transition is a NIC transition. That is, for some $\mathrm{a} < b < e - 1$, $\pi_{[b]} \rightarrow_{CPU} \pi_{[b+1]} \rightarrow_{NIC} \pi_{[b+2]}$. Then, there exists an exception handler sub-execution trace $\pi'\ [a{:}e]$ such that the order of the operations described by $\pi_{[b]} \rightarrow_{CPU} \pi_{[b+1]}$ and $\pi_{[b+1]} \rightarrow_{NIC} \pi_{[b+2]}$ are reversed. That is, $\pi'_{[b]} \rightarrow_{NIC} \pi'_{[b+1]} \rightarrow_{CPU} \pi'_{[b+2]}$. In addition, this is the only difference between $\pi[a{:}e]$ and $\pi'\ [a{:}e]$. Formally:

$$\begin{array}{l}
\forall \pi \in RM.\Pi,\ i \in IM.S,\ a,\ e,\ a < b < e - 1. \\
\quad \pi_{[a]} \rightsquigarrow_{real} \pi_{[e]}\ \wedge \\
\quad CPUL(\pi_{[a]})\ \wedge\ CPUL(\pi_{[e]})\ \wedge\ EHT(\pi,\ a,\ e)\ \wedge \\
\quad \pi_{[a]}\ R\ i\ \wedge \\
\quad label(\pi,\ b) = CPU\ \wedge\ label(\pi,\ b{+}1) = NIC \\
\quad \Rightarrow \\
\quad \exists \pi' \in RM.\Pi. \\
\qquad \pi'_{[a]} \rightsquigarrow_{real} \pi'_{[e]}\ \wedge\ label(\pi,\ b) = NIC\ \wedge\ label(\pi,\ b{+}1) = CPU\ \wedge \\
\qquad \pi[a{:}b] = \pi'\ [a{:}b]\ \wedge\ \pi[b{+}2{:}e] = \pi'\ [b{+}2{:}e].
\end{array}$$

- Exceptions Preserve R Lemma.

Assume $r\ R\ i$ and $CPUL(r)$ hold for a real state r in $RM.S$ and an ideal state i in $IM.S$, and the next CPU instruction execution from r causes the CPU to take an exception. Then the next CPU instruction execution from i also causes the CPU to take an exception, which is identical to the one taken from r, and the real state and the ideal state following the exceptions are related by R. Formally:

$$\begin{array}{l}
\forall r,\ r' \in RM.S,\ i \in IM.S. \\
\quad r\ R\ i\ \wedge\ CPUL(r)\ \wedge\ r \rightarrow_{EXC} r' \in \mathrm{RM}.\delta \\
\quad \Rightarrow \\
\quad \exists i' \in IM.S.\ i \rightarrow_{EXC} i'\ \wedge\ r'\ R\ i'.
\end{array}$$

- NIC Preserves R Lemma.

Assume that a real state r in the real model and and ideal state i in the ideal model are related by R, and that a NIC transition starts from r and ends in the real state r'. Then there exists a NIC transition in the ideal model from i to an ideal state i' such that r' and i' are related by R. Formally:

$\forall r, r' \in RM.S, i \in IM.S.$
$r \; R \; i \wedge r \rightarrow_{NIC} r'$
$\Rightarrow$
$\exists i' \in IM.S. \; i \rightarrow_{NIC} i' \; \wedge r' \; R \; i'.$

- Exception Handlers Preserve R Lemma.

  Assume π' $[b{:}c]$ is an exception handler sub-execution trace in $RM.\Pi$ such that the following four conditions hold:

  - $\pi'_{[b]}$ is the state from which the exception handler execution starts. (This means that either $\pi'_{[b]}$ immediately follows the transition in π' in which the CPU takes the exception causing the following exception handler sub-execution trace π' $[b{:}c]$, or only NIC transitions occur in π' between that CPU exception transition and $\pi'_{[b]}$.)
  - $\pi'_{[c]}$ is the state to which the exception handler execution ends. (That is, $\pi'_{[c]}$ is the first state in π' after $\pi'_{[b]}$ which satisfies *CPUL*.)
  - No NIC transitions occur in π' $[b{:}c]$.
  - $\pi'_{[b]} \; R \; \nu_{[b']}$ holds, where ν is an execution trace in the ideal model.

  Then there exists a specification transition in the ideal model from $\nu_{[b']}$ to a state $\nu_{[b'+1]}$, $\nu_{[b']} \rightarrow_{SPEC} \nu_{[b'+1]}$, such that $\pi'_{[c]} \; R \; \nu_{[b'+1]}$ holds.

The last three preservation of *R* lemmas have the intuitive meaning that CPU transitions describing CPU exceptions, NIC transitions, and exception handler sub-execution traces in the real model can be matched with respect to R by CPU transitions describing CPU exceptions, NIC transitions, and specification transitions in the ideal model. These three lemmas are used to prove Lemma V for transitions $r \rightsquigarrow_{real} r'$ that correspond to single NIC transitions and exception handler sub-execution traces. The following three subsection describes how Lemma V can be proved for Linux transitions, NIC transitions and exception handler sub-execution traces.

6.4.4.2 Lemma V for Linux Transitions

Assume $r \rightsquigarrow_{real} r'$ corresponds to a Linux transition,

$$\exists \pi \in RM.\Pi, 0 \leq a < e < length(\pi).$$
$$r = \pi_{[a]} \wedge r' = \pi_{[e]} \wedge CPUL(\pi_{[a]}) \wedge CPUL(\pi_{[e]}) \wedge LT(\pi, a, e),$$

and that *r* is related by *R* to some ideal state *i* in *ILM.S*, *r R i*. The following is a motivation of why

$$\exists i' \in \mathit{ILM.S}.\ i \rightsquigarrow_{ideal} i' \wedge r'\ R\ i'$$

holds.

First, *SEC*(*i*) holds by Lemma II and the definition of *ILM.S* being a subset of *IM.S*. Second, the CPU models in the real model and the ideal model describe identical CPU behavior for non-privileged CPU execution, which immediately follows by the definitions of the real model and the ideal model. These two properties are needed in following reasoning.

If the virtual address of the program counter in the ideal state *i* is mapped to a block of type ⊥, *MN* or *N*, then *mmu*(*i*, *PL0*, *i.cpu.uregs.r15*, *ex*) = ⊥, since *LINUX*(*i*) and *NIC_READ_ONLY*(*i*) of *SEC*(*i*) imply that such blocks are mapped as not executable. Since *r R i* requires the program counter values in *r* and *i* to be equal, the *MMU lemma* implies that *mmu* returns identical values for *r* and *i*, and therefore *mmu*(*r*, *PL0*, *r.cpu.uregs.r15*, *ex*) = ⊥. This in turn implies that the CPU in the real model takes an exception and enters privileged mode in *r'*. This leads to a contradiction since *CPUL*(*r'*) is assumed, and therefore the virtual address of the program counter in *i* is mapped by *mmu* to a block of type *L1*, *L2* or *D*. According to *LINUX*(*i*), blocks of type *L1*, *L2* or *D* are allocated to Linux.

The CPU in the real model and the CPU in the ideal model executes identical instructions from *r* and *i*, since (i) the virtual addresses of the program counters in *r* and *i* are mapped to the same physical address in Linux memory, and (ii) *r R i* requires the contents of Linux memory to be equal in *r* and *i*. Furthermore, the execution of those two identical instructions will operate on identical data. There are two reasons. First, *r R i* requires the following registers to be equal in *r* and *i*: The general-purpose and CPSR CPU registers, and the NIC registers. Second, all memory operands are equal for the instruction executions (data in memory the instructions operate on) from *r* and *i*, since:

- The virtual address references made by the instructions are equal due to equal CPU registers and identical instructions.
- The virtual to physical address mappings are equal in *r* and *i* according to the MMU lemma.
- As reasoned above, the memory references cause no exceptions and address locations in Linux memory, which is equal in *r* and *i*.

The executed CPU instructions, described by $r \rightarrow_{CPU} r'$ and $i \rightarrow_{CPU} i'$, will therefore perform identical operations. The state components in *r* and *i* that are related by *R* are equal and modified identically to produce *r'* and *i'*. Hence, the state components that *R* depends on are also equal between *r'* and *i'*, and therefore *r' R i'* holds.

Finally, it must be shown that $i \rightsquigarrow_{ideal} i'$ holds. That is, it must be shown that *CPUL*(*i*) ∧ *CPUL*(*i'*) holds and $i \rightsquigarrow_{ideal} i'$ is equal to $v_{[a]} \rightsquigarrow_{ideal} v_{[e]}$ for an execution trace $v \in \mathit{IM.\Pi}$ such that *LT*(*v*, *a*, *e*) holds (see definition of *ILM.δ* in Subsection 6.4.2; *NT* and *EHT* are not relevant since the transition is a CPU transition involving no exception handler execution). *CPUL*(*i*) and *CPUL*(*i'*) hold since *CPUL*(*r*), *CPUL*(*r'*), *r R i* and *r' R i'* hold and *R* requires equality on CPSR and DACR between related states.

Since *i* is assumed to be in *ILM.S* and $ILM.S \subseteq IM.S$, *i* is in *IM.S*. The definitions of *IM.S* and *IM.Π* imply that all states in *IM.S* are visited by some execution trace in *IM.Π* (see their definitions in Subsection 5.3.3), and therefore *i* is visited in an execution trace in *IM.Π*. $i \rightarrow_{CPU} i'$ is described by one of the three CPU transition rules in Subsection 5.3.2.1, and by the definition of *IM.Π* there exists therefore an execution trace $v \in IM.\Pi$ that includes $i \rightarrow_{CPU} i'$. This means that there exists indexes *a* and *e* of *v* such that $a + 1 = e$, $label(v, a) = CPU\ (= LT(v, a, e))$, $v_{[a]} = i$ and $v_{[e]} = i'$. Hence, $i \rightsquigarrow_{ideal} i'$ holds.

To conclude, $\exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \land r' \mathrel{R} i'$ holds.

6.4.4.3 Lemma V for Exception Handler Transitions

Assume $r \rightsquigarrow_{real} r'$ corresponds to an exception handler transition,

$$\exists \pi \in RM.\Pi,\ 0 \le a < e < length(\pi).$$
$$r = \pi_{[a]} \land r' = \pi_{[e]} \land CPUL(\pi_{[a]}) \land CPUL(\pi_{[e]}) \land EHT(\pi, a, e),$$

and that *r* is related by *R* to some ideal state *i* in *ILM.S*, *r R i*. The following is a motivation of why

$$\exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \land r' \mathrel{R} i'$$

holds.

Assume $r \rightsquigarrow_{real} r'$ corresponds to a sub-execution trace $\pi[a{:}e]$, $\pi \in RM.\Pi$. Since $\pi[a{:}e]$ describes an execution of an exception handler, it might involve CPU transitions describing executions of CPU instructions located in hypervisor or monitor memory, and possibly also NIC transitions. According to the definition of *EHT*, $\pi[a{:}e]$ has the shape:

$$\pi[a{:}e] = \pi_{[a]} \rightarrow_{EXC} \pi_{[a+1]} \rightarrow \ldots \rightarrow \pi_{[c\text{-}1]} \rightarrow_{RET} \pi_{[c]} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \pi_{[e]},$$

where the absence of a label on a transition between $\pi_{[a+1]}$ and $\pi_{[c\text{-}1]}$ denotes a transition of any type (exception return to start the execution of the monitor, exception to resume the execution of the hypervisor, execution of CPU instructions located in hypervisor or monitor memory, or NIC transitions). (According to the definition of *EHT*, the trailing NIC transitions do not necessarily occur but are included in the reasoning, since the case with no trailing NIC transitions is similar but simpler.)

To prove $\exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \land r' \mathrel{R} i'$, there are two cases to consider: Exception handlers that do not access NIC registers, and exception handlers that do access NIC registers. The latter case only occur when a NIC register write request handler is executed and is significantly more complex that the former case. This subsection reasons how Lemma V can be proved in the former case and the end of this subsection gives an intuitive view of how the latter case is handled. If the execution of an exception handler invokes the monitor, exception returns and exceptions will occur before and after the monitor is invoked. In the real model and in the ideal model, transitions describing exceptions and exception returns have the labels *EXC* and *RET*, respectively. In this subsection, such transitions have the label *CPU* if they occur before and after the execution of the monitor.

Assume no NIC register accesses are performed by the operations described by $\pi[a{:}e]$. The following is a reasoning of that there exists a transition $i \rightsquigarrow_{ideal} i'$ in $ILM.\delta$ such that $r'\ R\ i'$ holds and $i \rightsquigarrow_{ideal} i'$ is equal to $v_{[a']} \rightsquigarrow_{ideal} v_{[e']}$ for an execution trace $v \in IM.\Pi$. Furthermore, $EHT(v, a', e')$ holds for some indexes a' and e' and $v[a'{:}e']$ has the following shape:

$$v[a'{:}e'] = v_{[a']} \rightarrow_{EXC} v_{[a'+1]} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_{[c'-1]} \rightarrow_{SPEC} v_{[c']} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_{[e']}.$$

That is, first an exception occurs, then a consecutive sequence of NIC transitions followed by a specification transition that returns the CPU to Linux. Finally, another consecutive sequence of NIC transitions occur.

The proof of that v exists is made by constructing v, but before v is constructed, another sub-execution trace $\pi' \in RM.\Pi$ is constructed. The transitions in π' describe the same operations as described by the transitions in π, but the operations described by the transitions in $\pi'[a{:}e]$ might occur in a different order as described by the transitions in $\pi[a{:}e]$. In particular, the CPU transitions in $\pi'[a{:}e]$ describing the execution of an exception handler occur in consecutive sequence without being intermingled with NIC transitions. It can therefore be proved that the transitions in $\pi'[a{:}e]$ that describe the operations of an exception handler are identical to the operations described by some specification transition in $IM.\delta$. That is, the implementation of the hypervisor and the monitor operate according to the formal software design. π' is constructed by applying the *CPU and NIC Rescheduling Lemma* on $\pi[a{:}c]$ and then repeatedly on the result, such that $\pi'[a{:}e]$ has the following shape:

$$\pi'_{[a]} \rightarrow_{EXC} \pi'_{[a+1]} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \pi'_{[b]} \rightarrow_{CPU} \ldots \rightarrow_{CPU} \pi'_{[c\text{-}1]} \rightarrow_{RET} \pi'_{[c]} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \pi_{[e]},$$

and $\pi'_{[a]} = \pi_{[a]}$ and $\pi'_{[e]} = \pi_{[e]}$. First an exception occurs, then a consecutive sequence of NIC transitions followed by a sequence of transitions describing an execution of an exception handler, and finally another consecutive sequence of NIC transitions.

The following describes how v can be constructed. As was reasoned at the end of Subsection 6.4.4.2, i is visited by some execution trace in $IM.\Pi$. Let v be such an execution trace such that $v[a'] = i$ for some a'. The transitions of v following $v_{[a']}$ are derived according to the following four steps (Figure 45 illustrates these steps graphically):

1. Since $r\ R\ i$ is assumed, where $r = \pi_{[a]} = \pi'_{[a]}$ and $i = v_{[a']}$, the *Exceptions Preserve R Lemma* can be applied on $\pi'_{[a]} \rightarrow_{EXC} \pi'_{[a+1]}$ and $v_{[a']}$ to derive the transition $v_{[a']} \rightarrow_{EXC} v_{[a'+1]}$ such that $\pi'_{[a+1]}\ R\ v_{[a'+1]}$ holds:

   $$v[a'{:}a'+1] = v_{[a']} \rightarrow_{EXC} v_{[a']}.$$

2. The *NIC Preserve R Lemma* can be applied on $\pi'_{[k']} \rightarrow_{NIC} \pi'_{[k'+1]}$ and $v_{[k]}$ such that $v_{[k']} \rightarrow_{NIC} v_{[k'+1]}$ and $\pi'_{[k+1]}\ R\ v_{[k'+1]}$ hold for $a < k < b$ and $a' < k' < c' - 1$, where $b - a = c' - 1 - a'$ (the same number of NIC transitions follow the exception transitions):

   $$v[a'{:}c'{-}1] = v_{[a']} \rightarrow_{EXC} v_{[a']} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_{[c'-1]}.$$

3. The *Exception Handlers Preserve R Lemma* can be applied on $\pi'[b{:}c]$ and $v_{[c'-1]}$ to derive the transition $v_{[c'-1]} \rightarrow_{SPEC} v_{[c']}$ such that $\pi'_{[c]}\ R\ v'_{[c']}$ holds:

   $$v[a'{:}c'{-}1] = v_{[a']} \rightarrow_{EXC} v_{[a']} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_{[c'-1]} \rightarrow_{SPEC} v_{[c']}.$$

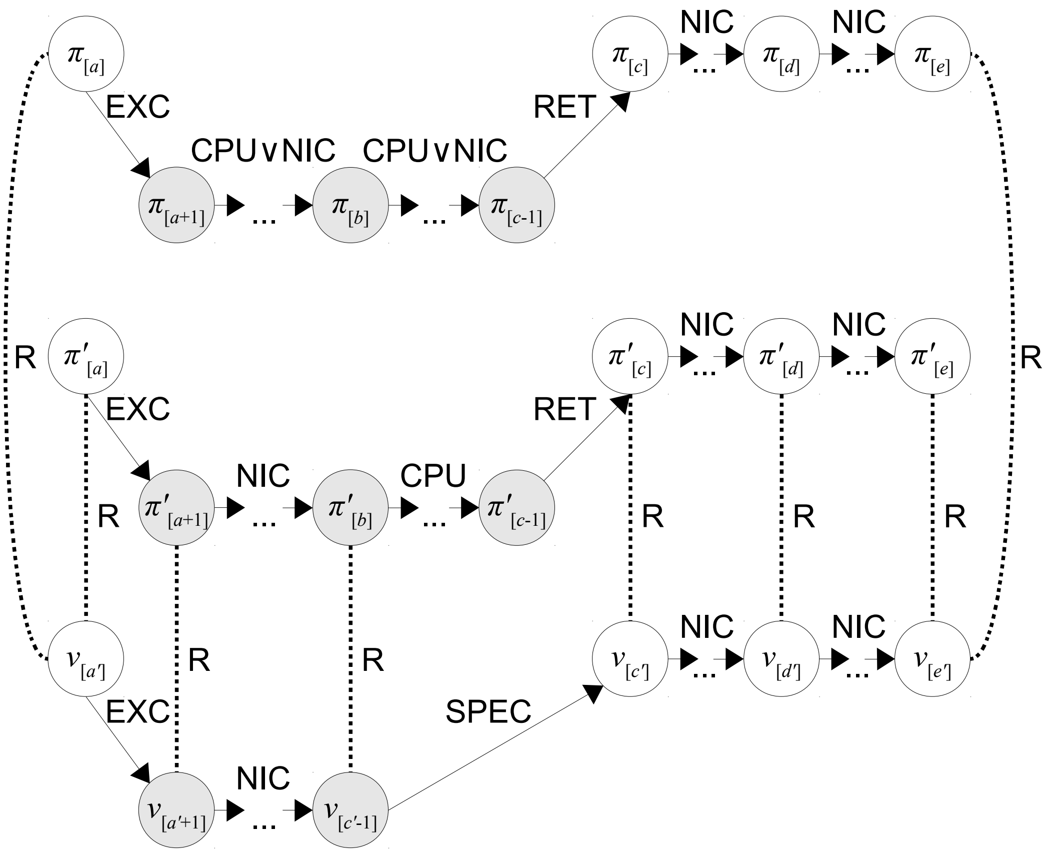


Figure 45: The relationship between π, π' and ν in the proof of Lemma V for exception handler transitions. No monitor invocations are shown. White states satisfy CPUL and shaded states do not satisfy CPUL. By assumption $\pi_{[a]}$ R $\nu_{[a']}$ holds. Since π[a:e] and π'[a:e] start from the same state and end in the same state, $\pi_{[a]} = \pi'_{[a]}$, $\pi_{[e]} = \pi'_{[e]}$ and $\pi'_{[a]}$ R $\nu_{[a']}$ holds. The application of the Exceptions Preserve R Lemma in step 1 gives $\pi'_{[a+1]}$ R $\nu_{[a'+1]}$. The applications of the NIC Preserve R Lemma in step 2 give $\pi'_{[b]}$ R $\nu_{[c'-1]}$. The application of the Exception Handlers Preserve R Lemma in step 3 gives $\pi'_{[c]}$ R $\nu_{[c']}$. The additional applications of the NIC Preserve R Lemma in step 4 give $\pi'_{[e]}$ R $\nu_{[e']}$. Since $\pi_{[e]} = \pi'_{[e]}$, $\pi_{[e]}$ R $\nu_{[e']}$ holds.

4. The *NIC Preserve R Lemma* can be applied on $\pi'_{[k']} \rightarrow_{NIC} \pi'_{[k'+1]}$ and $\nu_{[k]}$ such that $\nu_{[k']} \rightarrow_{NIC} \nu_{[k'+1]}$ and $\pi'_{[k+1]}$ *R* $\nu_{[k'+1]}$ hold for $c \leq k < e$ and $c' \leq k' < e'$, where $e - c = e' - c'$:

$$\nu[a'{:}e'] = \nu_{[a']} \rightarrow_{EXC} \nu_{[a']} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \nu_{[c'-1]} \rightarrow_{SPEC} \nu_{[c']} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \nu_{[e']}.$$

Since $r' = \pi_{[e]} = \pi'_{[e]}$ and $\pi'_{[e]}$ *R* $\nu_{[e']}$ hold, and by letting $\nu_{[e']} = i'$, r' *R* i' holds.

$i \rightsquigarrow_{ideal} i'$ holds since (see definition of *ILM.δ* in Subsection 6.4.2):

- All transitions in ν[a':e'] are described by the transition rules in Subsection 5.3.2, and therefore ν is in *IM.Π*. (See definition of *IM.Π* in Subsection 5.3.3.)

- $i = v_{[a']}$ and $i' = v_{[e']}$.
- $CPUL(v_{[a']})$ and $CPUL(v_{[e']})$ (and therefore i' is in *ILM.S*) hold because $CPUL(r)$, $CPUL(r')$, $r\ R\ i$ and $r'\ R\ i'$ hold and R requires equality of CPSR and DACR between related states, respectively (and the equalities in the previous bullet item).
- The sub-execution trace $v[a'{:}e']$ satisfies $EHT(v, a', e')$.
- $CPUL(v_{[e']})$ and $i' = v_{[e']}$ imply that i' is in *ILM.S*.

Hence, $\exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \wedge r'\ R\ i'$ holds for transitions $i \rightsquigarrow_{ideal} i'$ that correspond to sub-execution traces describing executions of exception handlers not accessing NIC registers.

Consider the case where an exception handler sub-execution trace accesses NIC registers. It has been reasoned that Lemma V holds for many sub-execution traces describing executions of the NIC register write request handlers (the only code accessing NIC registers). To save space and avoid repetition, Subsection B.4.2 provides a reasoning example of why Lemma V holds for sub-execution traces involving the execution of the most complex NIC register write request handler, namely *cppi_ram_handler* (described in Section 3.6). That example provides the main reasonings for proving Lemma V for exception handlers accessing NIC registers.

The main idea for proving Lemma V for exception handlers accessing NIC registers is the same as for exception handlers not accessing NIC register. Given an arbitrary sub-execution trace including NIC register accesses, find another sub-execution trace describing identical operations. The side conditions are that (i) each operation described by a NIC transition in the arbitrary sub-execution trace are described either before or after the execution of the exception handler in the other sub-execution trace, and (ii) the start and end states of the two sub-execution traces are equal, respectively.

As for exception handler sub-execution traces not accessing NIC registers, when finding such another sub-execution trace, operations described by a CPU transition and a NIC transition in the arbitrary sub-execution trace might occur in the opposite order in the other sub-execution trace. However, for exception handler sub-execution traces accessing NIC registers, the CPU instruction execution described by the CPU transition might access a NIC register which might affect or be affected by the NIC operations described by the NIC transition. Hence, the order of the operations described by the CPU transition and NIC transition affects the end state of the exception handler sub-execution traces. Dependent operations described by CPU and NIC transitions with such a relationship therefore cannot be reordered.

However, this does not mean that Lemma V cannot be proved for sub-execution traces involving an execution of a NIC register write request handler. Lemma V can be proved for sub-execution traces involving an execution of a NIC register write request handler by considering (i) all possible computation paths of each NIC register write request handler, and (ii) for each such computation path, all possible interleavings of CPU and NIC transitions that can possibly occur in the sub-execution trace describing the execution of that computation path. That is, all possible system execution scenarios are considered for an exception handler execution accessing NIC registers. Proving Lemma V for exception handlers accessing NIC registers in this way therefore requires a greater manual effort than for exception handlers not accessing NIC registers. However, many sub-execution traces describe identical operations (implying many sub-execution traces ending in the same state, denoted $\pi_{[e]}$ above), and therefore many sub-execution traces can be considered simultaneously.

Other details that must be considered when proving Lemma V for exception handlers accessing NIC registers are:

- NIC operations described by NIC transitions in π might have to be internally reordered in π', and NIC operations described by NIC transitions in π might in π' occur before the CPU transitions describing the execution of an exception handler while other NIC operations occur after the execution of the exception handler. (See Subsection B.4.2.)
- The recheduled order of NIC operations in π' might cause the execution of a NIC register write request handler to take another computation path compared to the execution of the NIC register write request handler in π. (See Subsection B.4.2.)
- When the CPU and the NIC access NIC registers that affect both the following execution of the CPU and the NIC, and how those accesses affect their following execution. This involved, for instance, accesses to the ownership and the end of queue bits of buffer descriptors and writes to the CP registers for asserting and deasserting interrupts.

Another interesting point is that the NIC register write request handlers can be designed to simplify the analysis by considering how the NIC device driver in Linux configures the NIC. For instance, before *cppi_ram_handler* performs any operations, it reads data structures of the software design in order to conclude that the NIC is not performing operations related to initialization or tear downs (*initialized*, *tx0_tearingdown* and *rx0_tearingdown* described in Section 3.4). *cppi_ram_handler* can do this because the NIC device driver in Linux never writes CPPI_RAM while the NIC performs initialization or tear down operations. However, this means that when Lemma V is proved for sub-execution traces involving *cppi_ram_handler*, no NIC transitions can occur that are described by the initialization, transmission teardown or the reception teardown automata. The NIC transitions that must be analyzed can therefore only be described by the transmission and reception automata, thereby simplifying the proof.

6.4.4.4 Lemma V for NIC Transitions

Assume $r \rightsquigarrow_{real} r'$ corresponds to a NIC transition,

$$\exists \pi \in RM.\Pi, 0 \leq a < e < length(\pi).$$
$$r = \pi_{[a]} \land r' = \pi_{[e]} \land CPUL(\pi_{[a]}) \land CPUL(\pi_{[e]}) \land NT(\pi, a, e),$$

and that r is related by R to some ideal state i in $ILM.S$, $r\ R\ i$. The following is a motivation of why

$$\exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \land r'\ R\ i'$$

holds.

The *NIC Preserves R Lemma* implies that there exists a NIC transition $i \rightarrow_{NIC} i'$ such that $r'\ R\ i'$ holds. $i \rightsquigarrow_{ideal} i'$ according to a reasoning similar to the one presented in the last two paragraphs in subsection 6.4.4.2, but with respect to the NIC transition rules in Subsection 5.3.2.2 and *NT*.

6.4.5 Lemma VI Implied by Lemma IV and Lemma V

This subsections describes how Lemma IV and Lemma V imply Lemma VI:

$$\forall r \in RLM.S.\ \exists i \in ILM.S.\ r\ R\ i.$$

This description relies on that each state $r \in RLM.S$ is visited by some execution trace $\psi \in RLM.\Pi$. Considering the definition of *RLM*, it is not obvious that this is true. The following is a reasoning of that each state $r \in RLM.S$ is visited by some execution trace $\psi \in RLM.\Pi$.

Let $r \in RLM.S$. By the definition of *RLM.S*, $r \in RM.S \land CPUL(r)$ holds. $r \in RM.S$ implies that r is visited by an execution trace $\pi \in RM.\Pi$ starting from $r_0 \in RM.IS$. For any state, the non-deterministic scheduler of the device model framework can always select the CPU model to describe the next transition from that state. There exists therefore an execution trace that visits r such that r is followed by a CPU transition. Let π be such an execution trace. That is, r is visited by π such that π includes a CPU transition starting from r.

By definition, $RLM.IS = RM.IS$, and therefore $r_0 \in RLM.IS$. According to the definitions of $RLM.\delta$ and $RLM.\Pi$, and since $r_0 \in RLM.IS$, π gives rise to a sequence of transitions in $RLM.\delta$ that constitutes an execution trace $\psi \in RLM.\Pi$. By the definition of $RLM.\delta$, each such transition corresponds either to a Linux transition, an exception handler transition or a NIC transition. Considering the definitions of these three types of transitions, *LT*, *EHT* and *NT*, only the transitions in ψ defined by *EHT* and *NT* can cause ψ to omit r.

Each transition in ψ defined by *EHT* corresponds to some sub-execution trace $\pi[a{:}e]$. *EHT* is defined in terms of *EH* and *TNT*, each describing one part of $\pi[a{:}e]$:

- *EH* describes the first part of $\pi[a{:}e]$, $\pi[a{:}c]$, $a < c \leq e$, which consists of transitions describing the execution of an exception handler. All of the states visited by $\pi[a+1{:}c-1]$ do not satisfy *CPUL* while $\pi_{[a]}$ and $\pi_{[c]}$ satisfy *CPUL*. Since r satisfies *CPUL*, r is not visited by $\pi[a+1{:}c-1]$.
- *TNT* describes the second part of $\pi[a{:}e]$, $\pi[c{:}e]$, $a < c \leq e$, which consists of NIC transitions, and where the transition from $\pi_{[e]}$ is a CPU transition. Since r is followed by a CPU transition in π, r is not visited by $\pi[c{:}e-1]$.

Since a transition defined by *EHT* starts from $\pi_{[a]}$ and ends in $\pi_{[e]}$, r is not omitted by ψ due to a transition defined by *EHT*.

The definition of *NT* prevents a NIC transition in π from being included in ψ if that NIC transition is a part of a consecutive sequence of NIC transitions immediately following an exception handler sub-execution trace. Such consecutive sequences of NIC transitions are described by *TNT* in the definition of *EHT*. As concluded in the second item bullet above, r is not visited by such a NIC transition sequence except possibly for the last state in such a sequence. Since such a last state is the end state of a transition defined by *EHT*, r is not omitted by ψ due to a transition defined by *NT*.

Since r was arbitrarily chosen, each state $r \in RLM.S$ is visited by some execution trace $\psi \in RLM.\Pi$:

$$\forall r \in RLM.S.\ \exists \psi \in RLM.\Pi,\ 0 \leq k \leq length(\psi).\ \psi_{[k]} = r.$$

Lemma VI can be derived as follows. Let $r \in RLM.S$. Following the previous reasoning, there exists an execution trace $\psi = \psi_{[0]} \rightsquigarrow_{real} \ldots \rightsquigarrow_{\mathrm{real}} \psi_{[k]} \in RLM.\Pi$, for some $0 \leq k \leq length(\psi)$, such that $r = \psi_{[k]}$, and where $\psi_{[0]} \in RLM.IS$ by the definition of $RLM.\Pi$. Implied by Lemma IV ($\forall r \in RLM.IS.\ \exists i \in ILM.IS.\ r\ R\ i$), there exists an ideal state $\omega_{[0]} \in ILM.IS$ such that $\psi_{[0]}\ R\ \omega_{[0]}$ holds. Applying Lemma V on $\psi_{[j]} \rightsquigarrow_{real} \psi_{[j+1]} \wedge \psi_{[j]}\ R\ \omega_{[j]}$ derives $\omega_{[j]} \rightsquigarrow_{ideal} \omega_{[j+1]} \wedge \psi_{[j+1]}\ R\ \omega_{[j+1]}$ such that $\omega_{[j+1]} \in ILM.S$, $0 \leq j < k$. Hence, $\psi_{[k]}\ R\ \omega_{[k]}$ and $\omega_{[k]} \in ILM.S$ hold. Since $r = \psi_{[k]}$ and by letting $i = \omega_{[k]}$, $r\ R\ i$ is derived, and where $i \in ILM.S$. Since r was chosen arbitrarily the desired conclusion is derived:

$$\forall r \in RLM.S.\ \exists i \in ILM.S.\ r\ R\ i.$$

6.5 Three Lemmas Implying Theorem I

This section covers the third part of the proof plan, briefly outlined in Subsection 6.2.2.3. The following three subsections describe how the final three lemmas, Lemma VII, VIII and IX, can be proved, respectively.

6.5.1 Lemma VII

Lemma VII states two properties:

- When the MMU maps the value of the program counter to a physical address allocated to Linux, then the CPU is configured to execute Linux (the CPU is in non-privileged mode and DACR[5:4] = 0b01).
- When the CPU is not configured to execute Linux, then the NIC is in a defined state.

Formally: $\forall r \in RM.S.\ [LCE(r) \Rightarrow CPUL(r)] \wedge [\neg CPUL(r) \Rightarrow \neg r.nic.dead]$.

It is first discussed how the first conjunct of Lemma VII can be proved. That predicate is false only if $LCE(r)$ is true and $CPUL(r)$ is false. All states in $RM.IS$ satisfy $CPUL$, since $RM.IS = RLM.IS \subseteq RLM.S$ and all states in $RLM.S$ satisfy $CPUL$ (see the definition of *RLM*). By definition of $RM.\Pi$, this means that the first state in an execution trace in $RM.\Pi$ satisfies $CPUL$. Since $CPUL$ only can be falsified by exceptions (only exceptions can modify the privilege level and DACR cannot be modified in non-privileged mode), states in $RM.S$ not satisfying $CPUL$ must follow an exception. An exception causes an exception handler of the hypervisor and the monitor to be executed. Hence, it is only the exception handlers of the hypervisor and the monitor that can break the first conjunct of Lemma VII, by setting the program counter to point to Linux memory when the CPU is not configured to execute Linux. That is, the CPU must be in non-privileged mode and

DACR[5:4] = 0b01 (*CPUL* is satisfied), when an execution of an exception handler sets the value of the program counter to be mapped to a physical address allocated to Linux (*LCE* gets satisfied).

To prove that all executions of all exception handlers configure the CPU to execute Linux when the program counter value is set to point to Linux memory, it suffices to prove that all exception handlers of the hypervisor and the monitor terminate correctly:

- All monitor invocations terminate.
- All executions of all exception handlers of the hypervisor terminate by sooner or later setting the program counter value to be mapped to the next Linux instruction to be executed, and simultaneously setting the CPU in non-privileged mode, at which point DACR[5:4] must be equal to 0b01.

These two termination properties imply that all sub-execution traces of exception handlers terminate in states satisfying *CPUL*, which is required by *EHT* in the definition of *RLM.δ*. All sub-execution traces of exception handlers therefore have a transition in *RLM.δ*. Since Lemma V states that each such transition in *RLM.δ* have a matching transition in *ILM.δ* with respect to *R*, these two termination properties and Lemma V imply that all exception handlers are correct with respect to the formal software design specified by the ideal model, *IM*.

The two termination properties can be proved to hold for the exception handlers as reasoned in the following three steps. First, assume that the last state *r* ∈ *RM.S* before an exception occurs is related to an ideal state *i* ∈ *IM.S*. This assumption can be made since each initial state in *RLM.IS* is related to some initial state in *ILM.IS*, according to Lemma IV, and Linux and NIC transitions do not break this relationship, according to Lemma V. This means that the last state *r* ∈ *RM.S* before the first exception occurs in an execution trace in *RM.Π* is related to some ideal state *i* ∈ *IM.S*. The termination properties then imply that the sub-execution trace of the invoked exception handler have a transition in *RLM.δ*. Lemma V therefore implies that the last state *r'* ∈ *RM.S* of the sub-execution trace of the invoked exception handler is related by *R* to some ideal state *i'* ∈ *IM.S*. This reasoning can then be applied inductively for later exception handler executions in an execution trace in *RM.Π*. This means that for each real state in *RM*, followed by an exception, there always exists some ideal state in *IM*, such that the two states are related by *R*.

Second, the *Constant Memory Lemma* states that a consecutive sequence of NIC transitions starting from an ideal state satisfying *SEC*, cannot modify the contents of the memory regions allocated to the hypervisor and the monitor. By Lemma II all states in *IM.S* satisfies *SEC*. Considering the ideal model *IM*, no NIC transitions from *i* can therefore modify the memory regions allocated to the hypervisor and the monitor. Since the operation of the NIC only depends on its own state, and *r R i* implies that the NIC is in the same state in *r* and *i*, considering the real model *RM*, no NIC transitions from *r* can modify the memory regions allocated to the hypervisor and the monitor. If all executions of the exception handlers never configures the NIC to write into hypervisor or monitor memory, the sub-execution traces of those executions can be considered in isolation without considering NIC transitions. The reason is that in this case, the termination properties of the exception handlers of the hypervisor and the monitor do not depend on the operation of the NIC, but only on the hypervisor and the monitor, since the NIC cannot modify the code nor the data of the hypervisor and the monitor.

Third, all exception handlers can be analyzed by generating their control flow graphs, as illustrated in Figure 46. Correct termination of the exception handlers can then be proved from properties implied by $r\ R\ i$ and $SEC(i)$ by proving that:

- All cycles in the control flow graph are finite.
- For each path of sufficient length, a control point in the exception handler is reached where the virtual address in the program counter is mapped to a physical address allocated to Linux. For the first such encountered control point in the considered path, the CPU must be in non-privileged mode with DACR[5:4] = 0b01.

This proof approach is similar to how the binary code of the exception handlers of a hypervisor was formally verified by Dam et al. [68] with BAP [80]. That approach is probably useful in this context as well.

As stated previously, this termination proof relies on the executions of the exception handlers of the hypervisor and the monitor to not configure the NIC to modify the memory regions of the hypervisor and the monitor. This property can be proved when proving the second conjunct of Lemma VII. The reason is that these two proofs both rely only on the exception handlers of the hypervisor and the monitor. (The second conjunct of Lemma VII depends only on the hypervisor and the monitor since $\neg CPUL$ is true only for states visited by exception handler executions.) These two proofs can be accomplished as follows.

As in the proof approach of the first conjunct of Lemma VII, assume that for a real state $r \in RM$ from which an exception occurs there exists some ideal state $i \in IM$ such that $r\ R\ i$ holds. $r\ R\ i$ and $SEC(i)$ (derived by means of Lemma II) imply that in r the data structures of the hypervisor and the monitor (described in Section 3.4) are consistent with the state of the NIC. By means of this information and by importing the control flow graphs of all exception handlers into HOL4, it can then be analyzed which values are written in which order to which NIC registers by the executions of those exception handlers. The HOL4 implementation of the NIC model can then be used to reason that the executions of the exception handlers do not configure the NIC such that the NIC might either (i) modify the memory regions allocated to the hypervisor or the monitor, or (ii) enter an undefined state.

6.5.2 Lemma VIII

Lemma VIII is

$$\forall r \in RM.S.\ CPUL(r) \Rightarrow \exists i \in IM.S.\ r\ R\ i.$$

Lemma VIII can be proved as follows. Let the real state r satisfy

$$r \in RM.S \land CPUL(r).$$

These two properties of r is in accordance with the definition of $RLM.S$,

$$RLM.S = \{r \mid r \in RM.S \land CPUL(r)\},$$

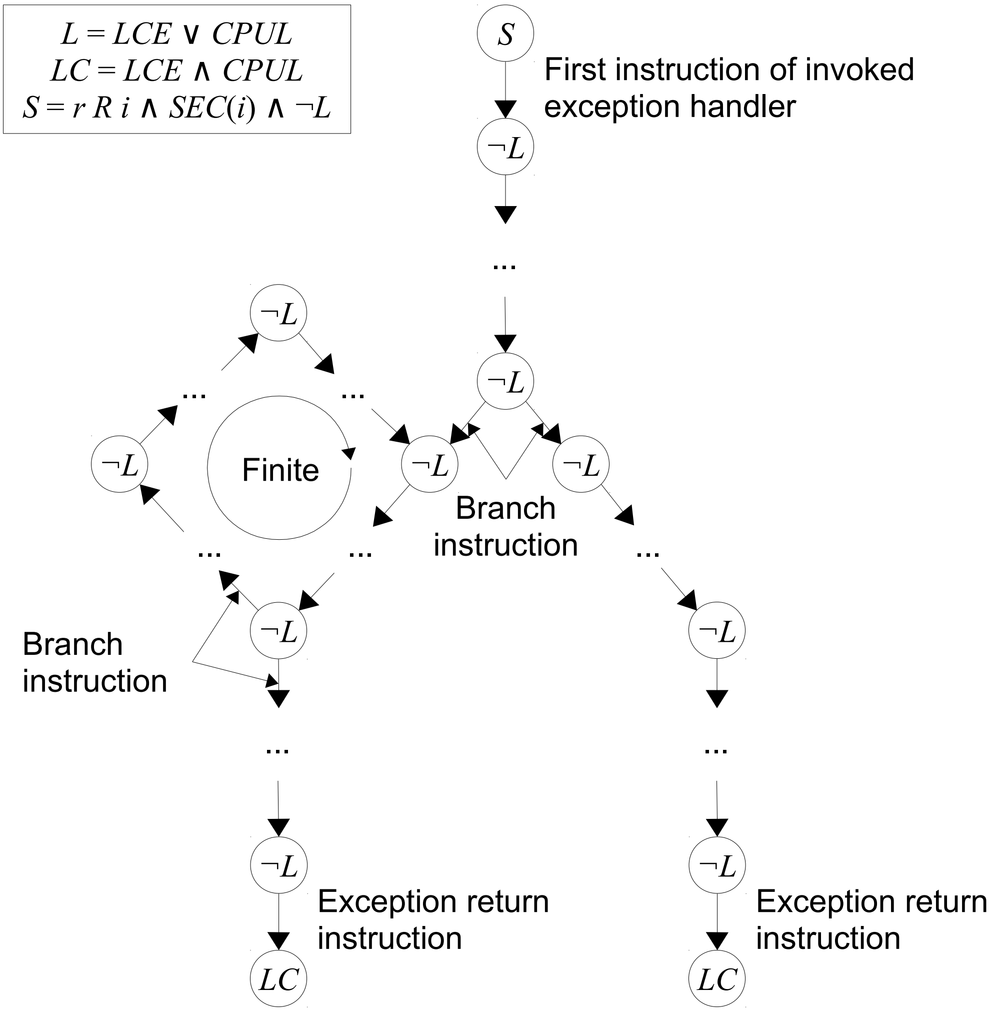

$L = LCE \vee CPUL$
$LC = LCE \wedge CPUL$
$S = r\ R\ i \wedge SEC(i) \wedge \neg L$
S
First instruction of invoked exception handler
$\neg L$
...
Finite
Branch instruction
Branch instruction
Exception return instruction
Exception return instruction
LC
LC

Figure 46: An example of a control flow graph of an exception handler. The graph includes control flow of the hypervisor and possibly also of the monitor. Each circle represents a control point in the exception handler, and each arrow represents a CPU instruction. The execution of the exception handler starts from a hardware state in which the CPU is in a state where it has just taken an exception. The real state representing that hardware state is assumed to be related to some state in the ideal model since CPU exceptions and NIC operations preserve R. Since all states in the ideal model satisfy SEC, that relationship enables certain properties of SEC to be transferred to the real state. The transferred properties are therefore assumed to hold at the first control point of the exception handler. Those assumptions are used to prove by means of the control flow graph that the exception handler terminates correctly. For all computation paths, two properties must be proved. First, executions of loops represented by cycles terminate. Second, a control point is reached at which both LCE and CPUL hold, neither of which hold at any previous control point.

and thus $r \in RLM.S$. Applying Lemma VI ($\forall r \in RLM.S.\ \exists i \in ILM.S.\ r\ R\ i$) gives $\exists i \in ILM.S.\ r\ R\ i$. Since $ILM.S$ is defined as a subset of $IM.S$,

$$ILM.S = \{i \mid i \in IM.S \wedge CPUL(i)\},$$

$\exists i \in IM.S.\ r\ R\ i$ holds. This proves Lemma VIII since r was chosen arbitrarily.

6.5.3 Lemma IX

Lemma IX is

$$\forall r \in RM.S, i \in IM.S.$$
$$r\ R\ i \Rightarrow [LCE(r) \Rightarrow LINUX\_CODE\_SIGNED(r)] \wedge \neg r.nic.dead.$$

Lemma IX can be proved as follows. Let $r \in RM.S$ and $i \in IM.S$ and assume $r\ R\ i$. Lemma III gives $\neg i.nic.dead$. Since $r\ R\ i$ implies $r.nic = i.nic$, $\neg r.nic.dead$ holds.

To prove $LCE(r) \Rightarrow LINUX\_CODE\_SIGNED(r)$, $LCE(r)$ is assumed, which reduces the proof to $LINUX\_CODE\_SIGNED(r)$. $LINUX\_CODE\_SIGNED(r)$ is by definition equal to

$$\forall code \in \text{<word32768>}.$$
$$code \in linux\_code(r) \Rightarrow sign(code) \in hvm\_to\_spec(r.memory).GI.$$

Assume $code \in linux\_code(r)$, where $code$ is an arbitrary bitstring of length 32678 (4 kB). The proof reduces to

$$sign(code) \in hvm\_to\_spec(r.memory).GI.$$

The *MMU Lemma* and $r\ R\ i$ (the content of Linux memory is the same in r and i) imply $linux\_code(i) = linux\_code(r)$, and therefore $code \in linux\_code(i)$ holds.

Recall that $LCE(r)$ is assumed, which by definition is equal to

$$\exists pa \in LINUX\_MEM.$$
$$mmu(r, PL0, r.cpu.uregs.r15, ex) = pa \vee mmu(r, PL1, r.cpu.uregs.r15, ex) = pa.$$

Since $r\ R\ i$ holds, $r.cpu.uregs.r15 = i.cpu.uregs.r15$. The *MMU lemma* can therefore be applied to derive:

$$\forall pl \in \{PL0, PL1\}.$$
$$mmu(r, pl, r.cpu.uregs.r15, ex) = mmu(i, pl, i.cpu.uregs.r15, ex).$$

Hence, $LCE(i)$ holds (recall that LCE is defined similarly for real states and ideal states).

Applying Lemma III again gives $EXEC\_SIGNED\_LINUX\_CODE(i)$, which by definition is equal to

$$LCE(i) \Rightarrow CPUL(i) \wedge LINUX\_CODE\_SIGNED(i).$$

Since $LCE(i)$ holds, $LINUX\_CODE\_SIGNED(i)$ holds, which by definition is equal to

$$\forall code \in \text{<word32768>}.\ code \in linux\_code(i) \Rightarrow sign(code) \in i.spec.GI.$$

Above, $code \in linux\_code(i)$ was derived, and thus $sign(code) \in i.spec.GI$ holds. Finally, $r\ R\ i$ implies $hvm\_to\_spec(r.memory).GI = i.spec.GI$, and the desired conclusion can therefore be derived:

$$sign(code) \in hvm\_to\_spec(r.memory).GI.$$

That concludes the description of how all top-level lemmas in the proof plan can be proved.

6.6 Conclusion and Discussion

This section discusses correctness and practicalities of the proof plan and presents a summary of the proof plan.

6.6.1 Correctness of Proof Plan

Since the real model is a transition system (device model framework integrating the ARMv7 CPU model and the NIC model), the four tuple *RM* describes the real model relatively accurately on paper by means of the labeled transition system notation described in Subsection 2.1.2. The top-level lemmas, logical formulas and many functions have also be defined or described accurately. Since these four components provide the base of the proof plan, the overall structure of the proof plan is probably correct.

It has been reasoned how each of the nine top-level lemmas can be proved by considering non-trivial details related to (some reasonings are omitted from the thesis to save space and avoid repetition):

- The formal and detailed definition of *SEC* in Appendix E.
- The preservation of *SEC* by the critical extended memory mapping request handlers, which are formally described in Appendix B. These formal descriptions have been considered, which include the requirements of these handlers (that are checked before executions of the handlers perform operations affecting the state, such as writing the page tables or updating the critical data structures ρ_{wt}, ρ_{ex} and τ), and the operations performed by the executions of these handlers.
- The preservation of *SEC* by the critical NIC register write request handlers, also formally described in Appendix B, and how Lemma V can be proved for many execution scenarios involving executions of these handlers. The pseudocode of the NIC register write request handlers and the NIC model has been considered.
- How Lemma V can be proved for exception handler executions not accessing NIC registers by means of a few sub-level lemmas.
- Motivations of all sub-level lemmas (in Appendix F), and which take deeper details into account (compared to the top-level lemmas), like the operation of the MMU, the NIC and the formal definition of *SEC*.

All of these considerations increase the correctness of the proof plan.

There is a significant set of details that have not been considered (deliberately and perhaps also by mistake). All details, of course, cannot be considered when making the pen-and-paper proof plan, but only when implementing the proof plan in HOL4, which provides computer support. Due to the amount of code, it is not unlikely that errors exist in the definitions of the NIC model, the security invariant *SEC*, and the NIC register write request handlers, and in the formal descriptions of the extended memory mapping request handlers. Such errors are revealed by HOL4 when the proof plan is implemented.

6.6.2 Practicalities of Proof Plan

In addition to the concern of whether the proof plan is correct, another concern is whether it is practically feasible to implement the proof plan in HOL4. The proof approach of applying the simulation proof method on *RM*/*RLM* (the real model representing the implementation of the complete system, including the the implementation of the formal software design) and *IM*/*ILM* (the ideal model specifying the formal software design), helps in making it feasible to implement the proof plan in several respects:

- The verification of the software design is separated from the verification of the implementation. The first part of the proof plan (Lemma I-III) is devoted to verify the software design, while the second part is devoted to verify that the implementation operates according to the software design (Lemma IV-VI). (The third part of the proof plan, Lemma VII-IX, is devoted to prove some additional properties of the implementation in order to imply the goal, Theorem I.) Proving Theorem I directly on the real model without the ideal model is probably more difficult and time consuming, than proving Theorem I by means of the simulation proof method with the ideal model. The reason is that high-level design concepts and low-level hardware details must be considered simultaneously in the former case, while they can be considered separately in the latter case. This difference in difficulty of reasoning that Theorem I holds, is true both when constructing the proof plan with pen and paper, and when implementing the proof plan in HOL4. For the same reason, the separation of the verification of the software design and the verification of the implementation of the software design, makes the HOL4 code implementing the proof plan easier to organize and understand, extend to prove new properties, or change because the hardware or software has changed.
- The software design and the implementation of it (described by *IM* and *RM*, respectively) are clearly decoupled and the software design is clearly documented. This makes it easier to understand, extend and modify the software design, the implementation of the software design, the proof plan, and the implementation of the proof plan, compared to if the software design and the implementation of it were integrated (both described by *RM*, since the implementation of the software design not only implements the software design but also describes it, although at a lower abstraction level, namely at the binary code level). The pseudocode of the NIC register write request handlers and the implementation of them demonstrate the decoupling of the software design and the implementation of it. For instance, a set of operations might be specified by the pseudocode by means of a recursive function, while implemented by means of a while loop. Likewise, the comments accompanying the pseudocode of the NIC registers write request handlers demonstrate the clear documentation of the software design. The formal definition of *SEC* further clarifies the design by unambiguously specifying what properties the software design must have. In addition, the software design is useful both for the proof plan and

for the implementation. As mentioned in Chapter 4, the implementation of the NIC register write request handlers was guided by their pseudocode.

As mentioned in the previous subsection, the proof plan, and additional material omitted from the thesis, have considered the main aspects necessary to consider for proving that only signed Linux code is executed, where only the deeper details have been omitted. With these considerations in mind, it does not seem to be impossible to implement the proof plan in HOL4 because of the following reasons:

- Implementing the NIC model, the NIC register write request handlers, and the formal definition of *SEC* in HOL4 is relatively straightforward by following their formal definitions. (Recall that the NIC register write request handlers are a part of the ideal model for describing the formal software design).
- The original memory mapping request handlers are already specified in HOL4 [86], only needing a few extensions, which have been formally specified. These and the NIC register write request handlers constitute the critical parts of the formal software design, specified by the specification transitions in the ideal model.
- The other handlers, for instance related to interrupt and cache management, are not accessing critical resources that *SEC* depends on. Proving that *SEC* is preserved by these other handlers is therefore easier than for the memory mapping and NIC register write request handlers. If the implementation of these proofs in HOL4 are still demanding, those implementations could be omitted for practical reasons, and still not significantly decreasing the reliability of the formal proof.
- Proving that the executions of the binary code of the exception handlers not accessing NIC registers (all but the data abort exception handler) operate as the formal software design can be accomplished by means of BAP [68], as mentioned in Subsection 6.5.1. Whether BAP is useful for verifying the binary code of the data abort exception handler (which might invoke the NIC register write request handlers to access NIC registers) remains to be investigated, since BAP is an external tool not considering the NIC model.
- As can be concluded from the previous item bullet list and the previous subsection, the proof plan is well-structured, includes several important and non-trivial details, and has a significant trustworthiness.

In addition to these five aspects of whether the proof plan is feasible to implement in HOL4, the work behind the two theorems proved in HOL4 about the device model framework [85], described in Subsection 2.4.2, might also be useful for implementing the proof plan in HOL4. Those two theorems conclude the following two isolation properties between a hypervisor and the guests executed on top of the hypervisor under certain system configuration assumptions:

- Non-infiltration: The execution of a guest and the devices only accessing the memory of that guest cannot deduce any information about or be affected by the state of (i) I/O devices only accessing memory of other guests, (ii) other guests, and (iii) the hypervisor.

- Non-exfiltration: The execution of a guest and the devices only accessing the memory of that guest cannot affect the state of (i) I/O devices only accessing memory of other guests, (ii) other guests, and (iii) the hypervisor.

Non-infiltration is not wanted for the monitor because that property prevents the monitor from both reading Linux memory when deciding whether Linux code is signed or not, and when reading ρ_{wt}, ρ_{ex} and τ, which are all located in hypervisor memory.

For the system configuration which Linux is executed in, four of the six assumptions of the two theorems are true, since the hypervisor can configure the system such that the NIC:

- only accesses Linux memory.
- is configured to never enter an undefined state.
- never writes to its own NIC registers.
- is not configured by the CPU when the CPU is executing a guest (with respect to the relevant properties of the NIC: which memory accesses are performed by the NIC and whether the NIC is in an undefined state).

Since executions of the monitor must be able to read ρ_{wt}, ρ_{ex}, τ and Linux memory, and the NIC asserts interrupts related to memory accesses the NIC has performed, the other two of the six assumptions of the two theorems (guest isolation, and I/O devices perform either memory accesses or interrupts, but not both) are false.

These contradictions between the current system configuration and these two theorems do not mean that the work behind the two theorems is not usable for proving that only signed Linux code is executed. Significant parts of the HOL4 code of the proofs of these two theorems and the ideas for proving the two theorems in HOL4, might be possible to reuse by replacing the unsatisfiable assumptions with assumptions reflecting the current system configuration: When the CPU is executing the monitor, the CPU can only read ρ_{wt}, ρ_{ex}, τ and Linux memory outside monitor memory, and NIC interrupts do not affect memory contents. None of these operations directly affect whether signed Linux code is executed or not.

Allowing the monitor to read ρ_{wt}, ρ_{ex}, τ and Linux memory, and with the replacing assumptions, two new theorems are formulated and which state similar properties as the original two theorems but adjusted for the current system configuration:

- When the CPU is executing Linux, the CPU cannot deduce any information from the states of the hypervisor and the monitor (non-infiltration for Linux), nor can the CPU affect the states of hypervisor and the monitor or configure the NIC (non-exfiltration for Linux).
- When the CPU executes the monitor, the CPU can, apart from only affecting the state of the monitor (non-exfiltration for the monitor), only read ρ_{wt}, ρ_{ex}, τ and the memory region allocated to Linux (extended non-infiltration for the monitor).

The HOL4 implementation and the ideas behind it of the original two theorems might then be useful to prove the two new theorems. Such a result would probably be a useful lemma for proving that the transitions in the ideal model preserve *SEC*. For instance, such a lemma would be useful when proving the preservation of the following statements of the following predicates of *SEC*:

- *SOUND_PT*: The memory mappings of the page tables are secure.
- *CONST_PT*: The executions of Linux do not modify page tables.
- *SOUND_MMU*: The memory regions of the hypervisor and the monitor contain their specified code.
- *LINUX*: The blocks typed as ⊥ are either unmapped or mapped as inaccessible by the page tables.
- *NIC_READ_ONLY*: The page tables prevent the CPU when the CPU is executing Linux to (i) interpret NIC register contents as CPU instructions, (ii) write NIC registers affecting memory accesses, and (iii) access unaligned physical addresses at which NIC registers are located.
- *CANNOT_DIE*: The page tables do not allow the CPU when executing Linux to configure the NIC such that the NIC enters an undefined state.

The challenges for implementing the proof plan in HOL4 are probably:

- Proving Lemma V for the data abort exception handlers which invoke the NIC register write request handlers. This task must, at least partly, be performed in HOL4 since the operation of the NIC (described by an HOL4 implementation NIC model) must be taken into account. The practical tool BAP might therefore be of limited use when proving Lemma V for the data abort exception handler.
- Proving the sub-level lemmas. The drawback of the sub-level lemmas is that the reasonings behind them, presented in Appendix F, are less formal, compared to the top-level lemmas. These reasonings are therefore less useful for implementing the sub-level lemmas in HOL4.
- Proving that *SEC* is preserved by all transitions in the ideal model. This task must consider all predicates of *SEC*, all CPU instructions, all exception handler specifications, and all step functions of the NIC model. The number of functions specifying the operations described by all transitions in the ideal model is probably over 150 and some of those functions specify complex operations.
- Proving other necessary lemmas involving details not considered in the proof plan. Since there is a significant set of details to consider in the ideal model and the real model, proving these additional lemmas in HOL4 is probably non-trivial. In addition, since these details have not been considered in the proof plan, difficult problems might have been overlooked.

The conclusion is therefore that the implementation of the proof plan in HOL4 is a demanding and time consuming task, but not impossible. Hence, the proof plan has a practical value.

6.6.3 Summary

Two significant design decisions in the proof plan are inspired by Dam et al. [69]:

- How the formal software design of the hypervisor and the monitor is specified by the ideal model as a set of atomic exception handlers.
- How the simulation proof method can be applied to prove that the binary code of the hypervisor and the monitor is correct.

Section B.5 discusses why the exception handlers of the formal software design in the ideal model are atomic. This is a justified question since atomic exception handlers make the proof of Lemma V more difficult than if the exception handlers were non-atomic.

The following is a brief summary of why the proof plan has the structure it has (based on the contents in Subsections 2.3.4.2, 5.2.3, 6.6.1 and 6.6.2):

1. Proof tools other than theorem provers, such as static analyzers, code annotation tools and model checkers, are not practical in this context due to complexity of the system to reason about and the sorts of properties to prove. The proof plan is therefore designed to be implemented in a theorem prover.
2. Modeling is a complex and time consuming task, making it a practical necessity to reuse already implemented models. Since the device model framework describes the hardware at sufficient detail in the theorem prover HOL4, the proof plan is based on the device model framework instantiated with a HOL4 implementation of the NIC model (described in Section 5.1), which is the real model, denoted by *RM*.
3. Since the real model is accurately described on paper by the four tuple *RM*, by means of the transition system notation described in Subsection 2.1.2, the reasoning in the proof plan, describing how it can be proved that only signed Linux code is executed, focuses on the four tuple *RM*.
4. To make the proof plan well-structured, practically feasible to implement and understandable, the proof approach of the proof plan is based on the simulation proof method applied on the ideal (Linux) model, *IM*/*ILM*, and the real (Linux) model, *RM*/*RLM*. This proof approach enables the separation of:
    - the proof (on *IM*/*ILM*) of that the (formal) software design is correct, which involves high-level design concepts, and
    - the proof (on *RM*/*RLM*) of that the implementation of the (formal) software design is correct, which involves low-level hardware and binary code aspects.

High-level design concepts and low-level hardware and binary code aspects are therefore considered separately, contributing to the manageability of the proof plan and its implementation in HOL4.

7 Results

This chapter summarizes and motivates why the results of the work described in Chapters 3 through 6 are solutions to the four problems listed in the problem definition in Section 1.2.

Consider the first problem of providing an extension of the design of the memory mapping request handlers and a design of the NIC register write request handlers, such that the hypervisor, the monitor and Linux are securely separated and only signed Linux code is executed. Sections 3.5, 3.6 and 3.4 present the extended design of the memory mapping request handlers, the design of the NIC register write request handlers and the data structures these handlers operate on, respectively.

In the original design of the memory mapping request handlers not considering the NIC [86], the page tables are configured such that the CPU:

- when executing the monitor, can only (i) access monitor memory, (ii) read the writable and executable reference counters (ρ_{wt} and ρ_{ex}, respectively) and the block type data structure (τ) located in hypervisor memory, and (iii) read Linux memory.
- when executing Linux, can only access Linux memory but not write the page tables, which are located in Linux memory.

That is, without the NIC, the memory mapping request handlers provide secure separation between the hypervisor, the monitor and Linux. In addition, all Linux blocks mapped as executable to the CPU when executing Linux have a signature in the golden image, and such blocks are not mapped as writable. That is, without the NIC, the memory mapping request handlers ensure that only signed Linux code is executed.

The NIC can perform two operations that affect the separation between the hypervisor, the monitor and Linux and whether only signed Linux code is executed: Access memory and modify NIC registers. In the extended design of the memory mapping request handlers considering the NIC, the page tables (i) are not allocated in blocks writable by the NIC, (ii) do not map NIC registers affecting memory accesses as writable to the CPU when the CPU is in non-privileged mode, and (iii) are not located at physical addresses locating NIC registers. That is, in the presence of the NIC (and assuming that the NIC register write request handlers securely configure the NIC), the memory mapping request handlers preserve the secure separation between the hypervisor, the monitor and Linux. In addition, the page tables do not map blocks writable by the NIC as executable nor map NIC registers as executable. That is, in the presence of the NIC, the memory mapping request handlers preserve the property of only signed Linux code execution. This extension of the design relies on the writable NIC reference counter (ρ_{NIC} located in hypervisor memory; when the CPU is executing the monitor, the CPU cannot write but only read this data structure) and two additional block types (*MN* for the blocks addressing NIC registers that affect which memory accesses that are performed by the NIC, and *N* for the other blocks addressing NIC registers). The extended design of the memory mapping request handlers is provided by a formal description (see Appendix B).

In the design of the NIC register write request handlers, the NIC is only allowed to access memory blocks allocated to Linux and write blocks that do not contain page tables nor are mapped as executable. In addition, the NIC is never configured such that it can enter an undefined state according to the NIC model (which formally describes when the NIC enters an undefined state), preventing the NIC from performing unknown operations. That is, the NIC register write request handlers preserve the secure separation between the hypervisor, the monitor and Linux, and the signed Linux code execution property. This design relies on, among other data structures, the executable reference counter (ρ_{ex}) and the block type data structure (τ). The design of the NIC register write request handlers is provided in pseudocode (see Appendix B).

Consider the second and third problems of providing an implementation of the NIC register write request handlers in the hypervisor, and an extension of the hypervisor and Linux such that Linux has Internet access when executed on top of the hypervisor. Chapter 4 describes the implementation, which has been tested on BeagleBone Black to indeed give Linux restricted access to the NIC and access to the Internet.

Consider the fourth problem of providing a proof plan that describes how it can be formally proved in HOL4 that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed, and that is based on the device model framework [85]. Chapter 5 describes the models that are used in the proof plan, and Chapter 6 describes the proof plan.

The goal to prove of only signed Linux code execution is defined in terms of the real model. The real model is the device model framework instantiated with a model of the NIC. The device model framework is a transition system that describes how an ARMv7 CPU executes binary code, and can integrate I/O device models to form a formal model of a complete computer system. A model of the NIC is provided in pseudocode (see Appendix C), and which describes, in terms of a transition system, how the NIC on BeagleBone Black accesses memory and when the NIC enters an undefined state. The device model framework instantiated with a HOL4 implementation of the NIC model therefore formally describes in HOL4 how the hardware executes in which the hypervisor, the monitor and Linux are executed. Hence, the real model is suitable to use for formally proving that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed by the CPU in the hardware system of interest.

The proof plan is based on the simulation proof method applied on the real model and the ideal model. The real model describes the implementation, including both hardware and software aspects. The ideal model differs from the real model with respect to the description of the exception handlers and how they are executed. In the real model, the exception handlers consist of the binary code of the hypervisor and the monitor, and are described as being executed by the CPU instruction by instruction, possibly interleaved with NIC operations. In the ideal model, the exception handlers consist of mathematical functions, whose operations are described as being performed atomically without being interleaved with NIC operations. These mathematical functions constitute the formal specification of the binary code implementation of the hypervisor and the monitor. Some of these functions are defined in terms of the functions describing the memory mapping and NIC register write request handlers. The real model and the ideal model are formalized on paper by means of a classic notation for labeled transition systems.

The proof plan is mainly structured around nine top-level lemmas, Lemma I-IX. These lemmas are defined in terms of the formalizations of the real model and the ideal model and accurately described functions. Descriptions are provided of how each of these lemmas can be proved and how they are applied to prove that only signed Linux code is executed.

Lemma I and Lemma II are used to prove Lemma III. The critical part of these lemmas is the security invariant *SEC*, of which a formal definition is provided (see Appendix E). If a state satisfies *SEC*, only signed Linux code is executable in that state and all transitions from that state preserve *SEC*. *SEC* is then used to prove Lemma III: In the ideal model, only signed Linux code is executed. This means that the formal specification of the hypervisor and the monitor is correct.

Lemma IV and Lemma V are used to prove Lemma VI. The critical parts of these lemmas are a simulation relation and two additional models. The simulation relation implies that the operations performed from two related states are identical. The additional models, the real Linux model and the ideal Linux model, include only states from which Linux code can be executed and transitions between such states, enabling the application of the simulation proof method. The simulation proof method is then applied on the real Linux model and the ideal Linux model with respect to the simulation relation to prove Lemma VI: For each state in the real model from which Linux code is executed, there exists a state in the ideal model from which the same Linux code is executed. This means that the executions of the binary code of Linux are identical on the binary interfaces described by the real model and the ideal model.

Lemma VII, VIII and IX are used to prove that only signed Linux code is executed on the binary interface described by the real model. Lemma VII implies that the executions of the binary code of the exception handlers of the hypervisor and the monitor terminate correctly, and do not configure the NIC such that the NIC can enter an undefined state. Lemma VII might be possible to prove by means of BAP and HOL4. Lemma VII, VIII and IX are used to transfer the property of only signed Linux code execution of the ideal model, stated by Lemma III, to the real model, enabled by the identical Linux executions in the ideal model and in the real model, stated by Lemma VI. This means that the binary code implementation of the hypervisor and the monitor is correct with respect to their formal specification.

Hence, a relatively accurate pen-and-paper description has been given for how it can be proved in HOL4 that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed by the CPU in the system of interest.

All problems listed in the problem definition in Section 1.2 have thus been solved.

8 Conclusion and Discussion

This chapter discusses what the work described in this thesis can be used for and the practical meaning of it. Thoughts, reflections and future work are also included.

8.1 Use within PROSPER

The current implementation of the hypervisor and the monitor is complete with respect to the software design described in Chapter 3. The next step is to implement all models in HOL4:

- The NIC model: Implement the pseudocode of the NIC model in HOL4.
- The real model: Instantiate the device model framework with the HOL4 implementation of the NIC model.
- The ideal model: Make a copy of the real model and modify the CPU model such that the CPU model in privileged mode applies the functions specifying the exception handlers of the hypervisor and the monitor. This means that these specification functions must also be implemented in HOL4. The function specifying the supervisor call exception handler must apply functions specifying the extended memory mapping request handlers. This involves extending the functions specifying the original memory mapping request handlers [86] to consider the NIC, as described in Section 3.5 and Appendix B. The function specifying the data abort exception handler can be partly implemented by applying a HOL4 implementation of the pseudocode of the NIC register write request handlers.

Thereafter, the proof plan can be implemented in HOL4, with certain tasks accomplished by means of BAP. If these steps are performed, then it has been formally verified, at the CPU instruction abstraction level, that the binary code of the hypervisor and the monitor ensures that only signed Linux code is executed by the CPU on BeagleBone Black, with Linux having Internet access. This means that if the signature function of the monitor is suitably chosen and the golden image of the monitor is suitably initialized, then, with high reliability, no malicious Linux code is executed. This includes both code being part of the Linux kernel, device drivers, modules loaded on demand, and applications.

PROSPER has recently extended the implementation of the hypervisor and the monitor to enable updates of the golden image. This makes the system more useful since software can be updated and new software can be downloaded, and which can then be executed if the signatures of that new code are in the updated golden image. (Without this update capability, if new software is developed after the initial golden image is created, it would not be possible to execute that new software since the signatures of it would most likely not be in the initial golden image.) A new golden image is downloaded by means of a Linux application, and another Linux application is used to invoke a system call of the Linux kernel, which in turn invokes a hypercall of the hypervisor, and which in turn invokes the monitor. These invocations pass information to the monitor about the new golden image. The monitor implements SHA-256 to check the validity of the new golden image with respect to a secret key stored in monitor memory. If the new golden image is valid, the old golden image is replaced by the new golden image and otherwise not. This implementation requires verification of that the golden image is updated securely. A definition in HOL4 must therefore be made of which requirements that must be satisfied in order for an update to be performed of the golden image. If these two additional verification tasks are performed, then the result is a system that is useful in numerous applications, and that with reliability prevents attackers from causing execution of software that can cause damage or danger. Such a system is therefore suitable for numerous security critical applications. Considering the examples mentioned in Chapter 1, such a system might be useful for building both industrial control systems and smartphones.

Another potentially useful result mentioned at the end of Section 6.4.2 and briefly motivated in Section F.12 is bisimulation between the real (Linux) model and the ideal (Linux) model. If it is proved that in the ideal (Linux) model the executions of Linux cannot deduce any information about the state of the formal specification (denoted by *i.spec*), then the bisimulation result implies that the executions of Linux cannot deduce any information about the state of the hypervisor or the monitor, and that the NIC does not prevent the transfer of that information security property from the ideal (Linux) model to the real (Linux) model. This information security property is important if confidential data is stored in the memory of the hypervisor or the monitor, which is the case for the secret key stored in monitor memory and that is used to check the validity of updates of the golden image. Bisimulation has been formally proved in HOL4 by Dam et al. [69] with respect to one model describing how two programs are executed by separate CPUs and on another model describing how the same two programs are executed on top of a hypervisor by one CPU. This bisimulation result implies that the executions of the two programs are identical in either system configuration.

The techniques used in PROSPER and in the work described in this thesis can also be used to secure other hardware platforms running other operating systems. The results produced by PROSPER and the work described in this thesis therefore give insights to how embedded systems can be secured from a more general perspective, rather than specifically for embedded systems based on an ARMv7 CPU and Linux. Hence, these results are interesting to manufacturers and users of (security critical) embedded systems implemented by means of other hardware and software. Motivated by Chapter 1, there is significant amount of work left before important systems used by society have the desired security and reliability.

8.2 Meaning and Cost of Formal Verification

In general, the purpose of formal verification is to verify that a system has an important property. It is therefore important to consider how likely it is that a system actually has a verified property. Since formal verification is performed with respect to models, it is critical that the models correctly describe the system to verify and do not omit relevant behavior, in order for the verification to have a practical value. As discussed in Subsections 5.1.3 and 5.2.2, the models used in the work described in this thesis might contain bugs and do not describe all relevant details of the hardware (e.g. possible overspecification of the order of NIC operations, and omitted cache behavior). This means that formal verification with respect to these models might not be completely accurate. Still, the models contain a significant amount of details. Also, in order for formal verification to be feasible, the software design must be carefully prepared and it must be accurately implemented. This means that even if no formal verification is performed, the carefully prepared design and its accurate implementation themselves improve the correctness of the system compared to if no formal verification is planned.

Constructing completely accurate hardware models is extremely difficult due to the significant amount of non-trivial details, and the lack of information given from hardware specifications. To construct completely accurate hardware models, it would probably be necessary to use automatic tools that produce models from the hardware description language code specifying the implementation of the hardware. However, such models might be too detailed, causing the formal verification task to consider irrelevant details, and therefore be unnecessarily laborious.

There are other potential details that formal verification might overlook. Bugs might exist in: models, verification tools (e.g. theorem prover), assembler, linker, loader and hardware. Additional errors can occur during operation, such as breaking components and transient errors in hardware. For practical reasons, certain less critical software components might not be verified, such as interrupt handlers in this context of signed Linux code execution, leaving additional verification gaps. For these reasons, the expression “high reliability” is used above in the conclusion that only signed Linux code is executed.

If the hypervisor and the monitor were not to be formally verified, then the development would have consisted only of reading hardware specifications, designing the software, implementing the software, and testing the software. Considering the additional tasks related to formal verification, (i) if considered needed, addressing the potential errors mentioned in the previous paragraph, (ii) constructing models, (iii) devising a proof plan, and (iv) implementing the proof plan, significant efforts are needed to construct highly reliable systems by means of formal verification, and theorem proving in particular. Hence, constructing highly reliable computer systems is extremely time consuming, and thus extremely expensive.

8.3 Sustainable Development and Ethics

As has been reported in the media, the global environment is negatively affected by the modern life style people commonly have today. It has also been reported about peoples’ psychological unhealthiness. It is therefore relevant to consider sustainable development and ethics in the context of the work described in this thesis. Regarding sustainable development, three aspects are relevant to consider:

- Social sustainability: How can the work described in this thesis affect the needs of the world’s population? As discussed above, this work can be used to improve the reliability and security of certain critical systems used by society. This work can also be used to prevent malicious programs in people’s home appliances relieving people from worrying about malicious programs destroying or disclosing their private data. Hence, this work can be used to better satisfy the needs of the world's population.

- Ecological sustainability: How can the work described in this thesis affect the global environment? Since this work is intended to be used in electronic devices, this question is mostly relevant for manufacturers and consumers of those devices. Questions relevant for manufacturers are which raw materials are used, how those materials are retrieved, and whether those materials are recyclable. For consumers the main question is whether the devices are driven by renewable energy. However, the extra software overhead due to the hypervisor and the monitor might increase the power consumption, which is not a problem if the devices are driven by renewable energy. Economical sustainability is therefore not relevant for this work.
- Economical sustainability: Can the work described in this thesis make the economical development in industry negatively affect society and the global environment? To answer this question, it is considered what economical investments are made to build products with this work. It is important that companies do not buy products and services from subcontractors and other companies that exploit people and the environment. Such actions can negatively impact peoples' mental health, their economical situation, and the environment. The answer to this question is controlled by industry since these decisions do not depend on this work. Economical sustainability is therefore not relevant for this work.

The ethical aspects are partly discussed in the previous item bullet list. However, the intention of this work and its applicability within PROSPER is good: To protect society from harmful computer attacks, and to demonstrate how secure computer systems can be constructed. Two potential ethical aspects are considered. First, formal verification is a rigorous method to analyze systems and therefore can be used to find security flaws in systems. The information in this thesis might therefore provide knowledge to attackers of how they can analyze systems to find security flaws in systems they want to attack. This sounds far-fetched, but unlikely incidents have happened in the past.

Second, users of products built with the work presented in this thesis might not be correctly informed of what has been verified or what formal verification means. Formal verification of that the hypervisor and the monitor ensures that only signed Linux code is executed might easily be translated by sellers to that the system has been proved to completely resist viruses. Non-technical users might then believe that the system is completely correct and never crashes, which is not the case. Also, to minimize overhead, instead of an implementation in the monitor of a cryptographically secure hash algorithm, a simple signature scheme or virus scanner might be used. Such simpler implementations could give end users a false sense of security.

8.4 Future work

Section 8.1 describes some future work within PROSPER with respect to the work described in this thesis, and Section 8.2 mentions some potential issues to attack for decreasing the verification gap. This section summarizes some other interesting future work.

The current NIC model describes interrupts as occurring non-deterministically. The NIC model might have to be extended to describe interrupts as occurring deterministically in case some properties are desirable to prove that depend on interrupts. Section C.10 describes which NIC registers the NIC model must include in order to describe interrupts as occurring deterministically.

Currently, the hypervisor does not allow Linux to use the system DMA controller nor the USB controller, since these I/O devices can access memory and the hypervisor has no support for configuring these I/O devices. Similarly to the NIC, these I/O devices are also configured through registers mapped into the virtual address space. The hypervisor can therefore protect the registers of these I/O devices that affect memory accesses similarly to how the hypervisor protects the registers of the NIC that affect memory accesses. That is, the hypervisor configures the page tables to prevent the CPU from writing those registers when the CPU is in non-privileged mode. When a data abort exception occurs due to a write to those registers, the data abort exception handler checks whether the write is secure. If the write is secure, then it is re-executed and otherwise not. The software design of the NIC register write request handlers therefore illustrates how these additional I/O devices can be supported and can thus be used as a starting point for implementing support for them. Such support would allow Linux to access any kind of I/O device connected to the AM335x system-on-chips from Texas Instruments (the chip containing the CPU and all I/O device controllers on BeagleBone Black). If the Linux 3.10 kernel has support for the system DMA and USB controllers, then such a system with the hypervisor, the monitor and Linux would have any capabilities as any other generic embedded system has. Such a system would then be useful in numerous applications making it competitive.

It is also interesting to port the hypervisor to other development boards, such as the Raspberry Pi boards, which also use ARMv7 CPUs. In the ARM architecture, registers of I/O devices are accessed by means of CPU instructions reading from or writing to the virtual address space. Hence, the hypervisor can support I/O devices accessing memory on other development boards similarly to how the hypervisor supports the NIC on BeagleBone Black.

References

[1] Cisco. The Internet of Things: How the Next Evolution of the Internet Is Changing Everything. Cisco; 2011.

[2] Ericsson [Internet]. Ericsson. 2016 [accessed 29 February 2016]. Available from: http://www.ericsson.com/thecompany/press/releases/2015/06/1925907

[3] Huawei. Global Connectivity Index: Benchmarking Digital Economy Transformation 2015. 2015.

[4] Zetter K. It's Insanely Easy to Hack Hospital Equipment [Internet]. WIRED. 2014 [accessed 11 February 2016]. Available from: http://www.wired.com/2014/04/hospital-equipment-vulnerable

[5] Recorded Future. Up and to the Right: ICS/SCADA Vulnerabilities by the Numbers. Recorded Future; 2015.

[6] Kovacs E. Russian Hackers Target Industrial Control Systems: US Intel Chief [Internet]. Securityweek. 2015 [accessed 11 February 2016]. Available from: http://www.securityweek.com/russian-hackers-target-industrial-control-systems-us-intel-chief

[7] Cyberattack on German Steel Plant Caused Significant Damage: Report [Internet]. Securityweek. 2014 [accessed 12 February 2016]. Available from: http://www.securityweek.com/cyberattack-german-steel-plant-causes-significant-damage-report

[8] Prokupecz S, Kopan T, Moghe S. Former official: Iranians hacked into New York dam [Internet]. CNN. 2015 [accessed 12 February 2016]. Available from: http://edition.cnn.com/2015/12/21/politics/iranian-hackers-new-york-dam

[9] Chatham House. Cyber Security at Civil Nuclear Facilities: Understanding the Risks. Chatham House; 2015.

[10] Greenberg A. Hackers Remotely Kill a Jeep on the Highway—With Me in It [Internet]. WIRED. 2015 [accessed 11 February 2016]. Available from: http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway

[11] Greenberg A. Hackers Reveal Nasty New Car Attacks – With Me Behind The Wheel [Internet]. Forbes. 2013 [accessed 11 February 2016]. Available from: http://www.forbes.com/sites/andygreenberg/2013/07/24/hackers-reveal-nasty-new-car-attacks-with-me-behind-the-wheel-video

[12] United States Government Accountability Office. Air Traffic Control: FAA Needs a More Comprehensive Approach to Address Cybersecurity As Agency Transitions to NextGen. United States Government Accountability Office; 2015.

[13] Current Android Malware [Internet]. Spreitzenbarth: Mobile Security and Forensics. 2011 [accessed 13 February 2016]. Available from: http://forensics.spreitzenbarth.de/android-malware

[14] Current iOS Malware [Internet]. Spreitzenbarth: Mobile Security and Forensics. 2011 [accessed 13 February 2016]. Available from: http://forensics.spreitzenbarth.de/android-malware

[15] Tech Specs [Internet]. Leikr. [accessed 11 February 2016]. Available from: http://www.leikr.com/tech-specs

[16] FAQ — ZoneMinder documentation [Internet]. Zoneminder. 2016 [accessed 7 March 2016]. Available from: http://zoneminder.readthedocs.org/en/latest/faq.html

[17] Linksys WRT54GL Wireless-G Wireless Router [Internet]. Linksys. 2016 [accessed 7 March 2016]. Available from: http://www.linksys.com/us/p/P-WRT54GL

[18] Vaughan-Nichols S. The five most popular end-user Linux distributions [Internet]. ZDNet. 2014 [accessed 11 February 2016]. Available from: http://www.zdnet.com/article/the-five-most-popular-end-user-linux-distributions

[19] Vaughan-Nichols S. CES 2015: The Linux penguin in your TV [Internet]. ZDNet. 2015 [accessed 11 February 2016]. Available from: http://www.zdnet.com/article/ces-2015-the-linux-penguin-in-your-tv

[20] Linux community touched by the touchscreen on Electrolux fridge [Internet]. Electrolux. 2010 [accessed 11 February 2016]. Available from: http://www.electroluxgroup.com/en/linux-community-touched-by-the-touchscreen-on-electrolux-fridge-8873

[21] Herold K. Using Linux in Medical Devices: What Developers and Manufacturers Need to Know [Internet]. Embedded Computing Design. 2012 [accessed 11 February 2016]. Available from: http://embedded-computing.com/white-papers/white-linux-medical-devices-developers-manufacturers-need-know

[22] GENIVI Alliance: FAQ [Internet]. Genivi. [accessed 11 February 2016]. Available from: http://www.genivi.org/faq

[23] Correspondent Banking [Internet]. Perto. [accessed 12 February 2016]. Available from: http://www.perto.com.br/en/solutions/correspondentbanking.html

[24] Brusehaver T. Linux in Air Traffic Control [Internet]. Linuxjournal. 2004 [accessed 7 March 2016]. Available from: http://www.linuxjournal.com/article/7066

[25] The Linux Foundation [Internet]. The Linux Foundation. [accessed 11 February 2016]. Available from: http://www.linuxfoundation.org

[26] Gallagher S. The Navy’s newest warship is powered by Linux [Internet]. Ars Technica. 2013 [accessed 11 February 2016]. Available from: http://arstechnica.com/information-technology/2013/10/the-navys-newest-warship-is-powered-by-linux

[27] SCADA as you've never seen it before [Internet]. Nuclear Engineering International. 2011 [accessed 11 February 2016]. Available from: http://www.neimagazine.com/features/featurescada-as-you-ve-never-seen-it-before

[28] open collaboration powers everything... [Internet]. The Linux Foundation. 2013 [accessed 11 February 2016]. Available from: http://www.linuxfoundation.org/homepage-slide/open-collaboration-powers-everything

[29] ARM Holdings plc. Annual Report 2015: Strategic Report. ARM Holdings plc; 2016.

[30] ARM Holdings plc. ARM Cortex - R Architecture: For Integrated Control and Safety Applications. ARM Holdings plc; 2013.

[31] BeagleBoard.org Foundation. BeagleBone Black System Reference Manual. BeagleBoard.org Foundation; 2014.

[32] Texas Instruments. AM335x Sitara™ Processors Technical Reference Manual. Texas Instruments; 2015.

[33] ARM Holdings plc. ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition. ARM Holdings plc.

[34] Peled D. Software reliability methods. New York: Springer; 2001.

[35] Almeida J, Frade M, Pinto J, Melo de Sousa S. Rigorous Software Development. London: Springer London; 2011.

[36] Baier C, Katoen J. Principles of Model Checking. Cambridge, Mass.: The MIT Press; 2008.

[37] ARM Holdings plc. ARM System Memory Management Unit Architecture Specification. ARM Holdings plc; 2016.

[38] AMD. AMD I/O Virtualization Technology (IOMMU) Specification. AMD; 2011.

[39] Intel. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. Intel; 2011.

[40] Guthrie F, Lowe S. VMware vSphere Design, 2nd Edition. John Wiley & Sons; 2013.

[41] Russinovich M, Solomon D, Ionescu A. Windows internals. Redmond, Wash: Microsoft Press; 2012.

[42] Xen Networking [Internet]. Xen Project wiki. 2016 [accessed 7 March 2016]. Available from: http://wiki.xenproject.org/wiki/XenNetworking

[43] Wind River. Wind River Hypervisor. Wind River; 2015.

[44] SYSGO. PikeOS Hypervisor. SYSGO; 2015.

[45] LynxSecure Major Capabilities [Internet]. Lynx Software Technologies. [accessed 7 March 2016]. Available from: http://www.lynx.com/products/secure-virtualization/lynxsecure-separation-kernel-hypervisor/lynxsecure-major-capabilities

[46] OKL4 Microvisor [Internet]. General Dynamics Mission Systems. [accessed 8 March 2016]. Available from: https://gdmissionsystems.com/cyber/products/trusted-computing-cross-domain/microvisor-products

[47] Masmano M, Ripoll I, Crespo A. XtratuM Hypervisor for LEON3 Volume 2: User Manual. 2011.

[48] Mentor Graphics. Mentor Embedded Hypervisor Datasheet. Mentor Graphics.

[49] Vasudevan A, Chaki S, Jia L, McCune J, Newsome J, Datta A. Design, Implementation and Verification of an eXtensible and Modular Hypervisor Framework. Security and Privacy (SP), 2013 IEEE Symposium on. Washington, DC: IEEE Computer Society; 2013.

[50] The Muen Separation Kernel [Internet]. The Muen Separation Kernel. [accessed 8 March 2016]. Available from: http://muen.codelabs.ch

[51] Menon A, Schubert S, Zwaenepoel W. TwinDrivers: Semi-automatic Derivation of Fast and Safe Hypervisor Network Drivers from Guest OS Drivers. Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY: ACM; 2009.

[52] Alkassar E, Hillebrand M, Knapp S, Rusev R, Tverdyshev S. Formal Device and Programming Model for a Serial Interface. 2004.

[53] Alkassar E, Hillebrand M. Formal Functional Verification of Device Drivers. Proceedings of the 2nd International Conference on Verified Software: Theories, Tools, Experiments. Berlin, Heidelberg: Springer-Verlag; 2008.

[54] Duan J. Formal Verification of Device Drivers in Embedded Systems [Ph.D]. The University of Utah; 2013.

[55] Kim M, Choi Y, Kim Y, Kim H. Formal Verification of a Flash Memory Device Driver – An Experience Report. 15th International SPIN Workshop on Model Checking of Software. Berlin, Heidelberg: Springer Berlin Heidelberg; 2008.

[56] Penninckx W, Mühlberg J, Smans J, Jacobs B, Piessens F. Sound Formal Verification of Linux's USB BP Keyboard Driver. NASA Formal Methods: 4th International Symposium. Berlin, Heidelberg: Springer Berlin Heidelberg; 2012.

[57] Monniaux D. Verification of Device Drivers and Intelligent Controllers: A Case Study. Proceedings of the 7th ACM & IEEE International Conference on Embedded Software. New York, NY: ACM; 2007.

[58] Alkassar E, Paul W, Starostin A, Tsyban A. Pervasive Verification of an OS Microkernel: Inline Assembly, Memory Consumption, Concurrent Devices. Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments. Berlin, Heidelberg: Springer-Verlag; 2010.

[59] Alkassar E, Schirmer N, Starostin A. Formal Pervasive Verification of a Paging Mechanism. Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer-Verlag; 2008.

[60] Starostin A, Tsyban A. Correct Microkernel Primitives. Electronic Notes in Theoretical Computer Science. 2008;217:169-185.

[61] Klein G, Andronick J, Elphinstone K, Murray T, Sewell T, Kolanski R et al. Comprehensive formal verification of an OS microkernel. ACM Transactions on Computer Systems. 2014;32(1):1-70.

[62] Klein G, Sewell T, Winwood S. Refinement in the Formal Verification of the seL4 Microkernel. Design and Verification of Microprocessor Systems for High-Assurance Applications. 2010;:323-339.

[63] The CompCert C compiler [Internet]. Compcert. [accessed 8 March 2016]. Available from: http://compcert.inria.fr/compcert-C.html

[64] Fernandez M, Klein G, Kuz I. Microkernel Verification Down To Assembly Extending the seL4 verification.

[65] Muen Separation Kernel Lays Open Source Foundation for High-Assurance Software Components [Internet]. Adacore. 2013 [accessed 8 March 2016]. Available from: http://www.adacore.com/press/muen-separation-kernel

[66] Moy Y. Muen Separation Kernel Written in SPARK [Internet]. Spark 2014. 2013 [accessed 8 March 2016]. Available from: http://www.spark-2014.org/entries/detail/muen-separation-kernel-written-in-spark

[67] Sanán D, Butterfield A, Hinchey M. Separation Kernel Verification: The Xtratum Case Study. Verified Software: Theories, Tools and Experiments. 2014;:133-149.

[68] Dam M, Guanciale R, Nemati H. Machine Code Verification of a Tiny ARM Hypervisor. Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices. New York, NY: ACM; 2013.

[69] Dam M, Guanciale R, Khakpour N, Nemati H, Schwarz O. Formal Verification of Information Flow Security for a Simple ARM-based Separation Kernel. Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. New York, NY: ACM; 2013.

[70] Dam M, Guanciale R, Nemati H. Trustworthy Virtualization of the ARMv7 Memory Subsystem. SOFSEM 2015: Theory and Practice of Computer Science: 41st International Conference on Current Trends in Theory and Practice of Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg; 2015.

[71] Dam M, Do V, Guanciale R, Nemati H, Vahidi A. Trustworthy Memory Isolation of Linux on Embedded Devices. 8th International Conference on Trust & Trustworthy Computing. Cham: Springer International Publishing; 2015.

[72] Alkassar E, Hillebrand M, Paul W, Petrova E. Automated Verification of a Small Hypervisor. Verified Software: Theories, Tools, Experiments: Third International Conference. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010.

[73] Paul W, Schmaltz S, Shadrin A. Completing the Automated Verification of a Small Hypervisor - Assembler Code Verification. Proceedings of the 10th International Conference on Software Engineering and Formal Methods. Berlin, Heidelberg: Springer-Verlag; 2012.

[74] Baumann C, Beckert B, Blasum H, Bormer T. Formal Verification of a Microkernel Used in Dependable Software Systems. Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security. Berlin, Heidelberg: Springer-Verlag; 2009.

[75] VCC: A Verifier for Concurrent C [Internet]. Microsoft Research. [accessed 9 March 2016]. Available from: http://research.microsoft.com/en-us/projects/vcc

[76] Jacobs B, Piessens F. The VeriFast Program Verifier. 2008.

[77] The CBMC Homepage [Internet]. The CBMC Homepage. [accessed 9 March 2016]. Available from: http://www.cprover.org/cbmc

[78] Verified Software Toolchain [Internet]. Verified Software Toolchain. [accessed 9 March 2016]. Available from: http://vst.cs.princeton.edu

[79] Appel A. Verified Software Toolchain. Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software. Berlin, Heidelberg: Springer-Verlag; 2011.

[80] Brumley D, Jager I, Avgerinos T, Schwartz E. BAP: A Binary Analysis Platform. Proceedings of the 23rd International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer-Verlag; 2011.

[81] Fehnker A, Huuck R, Rauch F, Seefried S. Some Assembly Required - Program Analysis of Embedded System Code. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE; 2008.

[82] Maus S. Verification of Hypervisor Subroutines written in Assembler [Ph.D]. Albert-Ludwigs-Universität at Freiburg; 2011.

[83] Li G. Formal Verification of Programs and Their Transformations [Ph.D]. The University of Utah; 2010.

[84] Fox A, Myreen M. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. Proceedings of the First International Conference on Interactive Theorem Proving. Berlin, Heidelberg: Springer-Verlag; 2010.

[85] Dam M, Schwarz O. Formal Verification of Secure User Mode Device Execution with DMA. Hardware and Software: Verification and Testing: 10th International Haifa Verification Conference. Cham: Springer International Publishing; 2014.

[86] Chfouka H, Dam M, Ekdahl P, Guanciale R, Nemati H. Trustworthy Prevention of Code Injection in Linux on Embedded Devices. Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security. Cham: Springer International Publishing; 2015.

[87] OCP-IP Association. Open Core Protocol Specification. OCP-IP Association; 2006 p. 9, 14, 17.

[88] Addison D. Embedded Linux applications: An overview [Internet]. 2001 [accessed 31 August 2016]. Available from: http://www.ibm.com/developerworks/library/l-embl

[89] McMillan S. Why choose Linux for embedded development projects? [Internet]. EETimes. 2002 [accessed 31 August 2016]. Available from: http://www.eetimes.com/document.asp?doc_id=1277902

[90] Processors [Internet]. arm.com. 2016 [accessed 31 August 2016]. Available from:
http://www.arm.com/products/processors

[91] Enabling a Standard [Internet]. arm.com. 2016 [accessed 31 August 2016]. Available from:
http://www.arm.com/about/our-story/enabling-a-standard.php

[92] Filliâtre J, Gondelman L, Paskevich A. The Spirit of Ghost Code. Proceedings of the 16th International Conference on Computer Aided Verification. New York, NY: Springer-Verlag New York; 2016.

[93] Xen ARM with Virtualization Extensions whitepaper [Internet]. Wiki.xen.org. 2016 [accessed 31 August 2016]. Available from: http://wiki.xen.org/wiki/Xen_ARM_with_Virtualization_Extensions_whitepaper

[94] The Netperf Homepage [Internet]. Netperf.org. 2016 [accessed 8 September 2016]. Available from:
http://www.netperf.org/netperf

[95] Raho M, Spyridakis A, Paolino M, Raho D. KVM, Xen and Docker: a performance analysis for ARM based NFV and Cloud computing.

Proceedings of the 2015 IEEE 3rd workshop on Advances in Information, Electronic and Electrical Engineering. IEEE; 2016.

Appendix A Pseudocode Notation

This appendix describes the pseudocode that is used to define the NIC model, NIC register write request handlers, functions used to define the top-level lemmas, and functions used to define *SEC*.

A.1 Numbers

Binary and hexadecimal numbers have the prefixes 0b and 0x. The underscore symbel '_' is used to ease the interpretation of long numbers, like 0x4A10_2000.

A.2 Sets and Types

A data type exactly the same meaning as a set. A variable that has a certain type contains a value that belongs to the set represented by that type. The primitive data types are:

- const: Denotes an arbitrary symbolic string. The declaration 'const: usr' makes the string 'usr' a symbolic constant which can be used for comparisons, for instance.
- nat $\overset{\text{def}}{=}$ {n | n ≥ 0}
- bool $\overset{\text{def}}{=}$ {false, true}
- wordx $\overset{\text{def}}{=}$ $\{0, 1\}^x$: Denotes set of all bit strings of length x.

Other types can be defined by means of already defined types. Such types are sets, functions, tuples, records, unions and lists. Let t1, ..., tn be already defined types, then these six types can be defined as:

- Set: 's = {t1}' means s $\overset{\text{def}}{=}$ {A | A ⊆ t1}. Let s = {nat}, s: v := {0, 1231, 559} means then that the variable v is of type {nat} and contains the numbers 0, 1231, and 559.
- Function: 'f = t1 → t2' means f: t1 → t2. Let f = word2 → word1, f: v means then that v takes one 2-bit string as argument and returns a single bit value. For instance:

  v(0b00) = 0b0, v(0b01) = 0b1, v(0b10) = 0b1, v(0b11) = 0.
- Tuple: 't = (t1, ..., tn)' means t $\overset{\text{def}}{=}$ t1×...×tn. Let t = (nat, bool), t: v means then that v has a natural number in its first component and boolean value in its second component. For instance, t: v := (7, false).
- Record: 'r = (t1: v1, ..., tn: vn)' means r $\overset{\text{def}}{=}$ t1×...×tn but with the ability to refer to a component by means of its variable name and the dot operator '.'. For instance, r = (bool: b, nat: n), r: v := (false, 3), implies v.b = false and v.n = 3.
- Union: 'u = t1 ∪ t2' means u $\overset{\text{def}}{=}$ t1 ∪ t2. Let u = nat ∪ {const: ⊥} and u: v. v can then not only be set to contain a natural number, but also be set to contain the symbol ⊥.

- List: 'l = [t1]' defines the type l as a list with elements of type t1. If l = [nat], then l: v := [0, 1, 2, 3] means that v is a variable that contains an ordered list of natural numbers.

A.3 Operators

A data type exactly the same meaning as a set. A variable that has a certain type contains a value that belongs to the set represented by that type. The primitive data types are:

- const: Denotes an arbitrary symbolic string. The declaration 'const: usr' makes the string 'usr' a symbolic constant which can be used for comparisons, for instance.
- nat $\overset{def}{=} \{n \mid n \geq 0\}$
- bool $\overset{def}{=}$ {false, true}
- wordx $\overset{def}{=} \{0, 1\}^x$: Denotes set of all bit strings of length x.

Other types can be defined by means of already defined types. Such types are sets, functions, tuples, records, unions and lists. Let t1, ..., tn be already defined types, then these six types can be defined as:

- Set: 's = {t1}' means s $\overset{def}{=} \{A \mid A \subseteq t1\}$. Let s = {nat}, s: v := {0, 1231, 559} means then that the variable v is of type {nat} and contains the numbers 0, 1231, and 559.
- Function: 'f = t1 → t2' means f: t1 → t2. Let f = word2 → word1, f: v means then that v takes one 2-bit string as argument and returns a single bit value. For instance:

  v(0b00) = 0b0, v(0b01) = 0b1, v(0b10) = 0b1, v(0b11) = 0.
- Tuple: 't = (t1, ..., tn)' means t $\overset{def}{=}$ t1×...×tn. Let t = (nat, bool), t: v means then that v has a natural number in its first component and boolean value in its second component. For instance, t: v := (7, false).
- Record: 'r = (t1: v1, ..., tn: vn)' means r $\overset{def}{=}$ t1×...×tn but with the ability to refer to a component by means of its variable name and the dot operator '.'. For instance, r = (bool: b, nat: n), r: v := (false, 3), implies v.b = false and v.n = 3.
- Union: 'u = t1 ∪ t2' means u $\overset{def}{=}$ t1 ∪ t2. Let u = nat ∪ {const: ⊥} and u: v. v can then not only be set to contain a natural number, but also be set to contain the symbol ⊥.

- List: 'l = [t1]' defines the type l as a list with elements of type t1. If l = [nat], then l: v := [0, 1, 2, 3] means that v is a variable that contains an ordered list of natural numbers.

A.4 Functions

The body S of pseudocode functions is defined by means of the following pseudocode statements:

- Variable declaration: 't: v1, … vn'. Declares the variables v1, …, vn to be of type t. The scope of the variables are with respect to the block in which they are declared.
- Variable declaration with assignment: 't: v := value'. nat: v := 0 initialized the variable v to zero. Another example is:

  '{const: linux, hypervisor, monitor}: software:= {linux}',

  which declares the variable software as being able to contain a set with the elements 'linux', 'hypervisor' and 'monitor', and is initialized to only contain the element 'linux'.
- Type definition: Is done as described above if the type is desired to be referred to by a single identifier, like t = (t1, ..., tn) for the tuple type with components whose values are of the types t1, ..., tn. The type can also be defined when the variable is declared as with {bool}: b := {true}.
- Assignment: 'v := exp'. Assigns the value of the expression exp to the variable v. Expressions are computed by means of the operators in the previous section.
- Assignment with operator: ':' op '='. The values of the left and right arguments are operated upon by the operation op ∈ {+, -, *, /, &, |} and the result is then assigned to the variable of the left argument. For instance, if word4: a = 0b1100 and word4: b = 0b0011, then the operation a :|= b will assign a the value 0b1111.
- Record component access: '.'. (See example in Section A.2 for record type definition.)
- Composition: One statement occurs after another by separating them by a new line.
- If statements: All sorts exists:
  - if condition then S
  - if condition then S else S1
  - if condition then S else if S1 ... else if S2
  - if condition then S else if S1 ... else Sn,

  where condition is a boolean expression and S, S1, S2 and Sn are statements generated by the rules in this list.
- Indentation: The block of an if statement is defined by indentation. For instance,

```
if a > b then
    c := 1
    d := 2
```

means that if a is greater than b, then are c and d assigned the values 1 and 2, respectively.

```
if a > b then
    c := 1
d := 2
```

means that if a is greater than b, then is c assigned the value 1. d is always assigned the value 2 irrespectively of the relationship between a and b.

- Returning function result 'return exp': The function with the return statement computes the value of the expression exp. Code following the return statement of the function is ignored.
- One line comments '//''. '//Comments' is a comment.
- Comments can also be surrounded by '/*' and '*/'. '/* This is a comment */' is a comment.

A function f with return type t and which takes n arguments of types t1, …, tn is defined as:

```
t: f(t1: arg1, ..., tn: argn)
    S
```

where the statement S is constructed according to the rules above. In classic mathematical notation: $f: t1 \times ... \times tn \rightarrow t$.

An example is:

```
nat: increment_function(nat: value)
    return value + 1

(nat, word32): example_function(nat: arg1, word32: arg2, word32: bits)
    if arg1 = 0 then
        return (0, arg2)
    else
        arg1 :+= increment_function(arg1)
    bits :|= bits << arg1
    return (arg1, bits)
```

Two functions are defined. increment_function return its argument incremented by one. example_function has three arguments with the types nat, word32 and word32, and returns a pair where the first component is a natural number and its second component a 32-bit bit string. If the first argument is zero, then example_function returns the pair with both components being equal to zero. Otherwise is the function increment_function applied on arg1 and increments arg1 with the returned value. Then it shifts bits left by the number of bit positions being equal to the content of arg1, performs bitwise OR with that left shift result and the current value of bits. That result is then stored in bits. Finally, arg1 and bits are returned as a pair. For instance, example_function(2, 0b10101010, 0b0110) returns (5, 0b011000110).

Appendix B Memory and NIC Handlers

This appendix describes some details that are related to the ideal model. The first section provides a formal definition of the ideal state, denoted as ideal_state. The second section describes the memory mapping request handlers in a more formal way, while the third section includes the pseudocode for the representative NIC register write request handlers and how they are applied when a data abort exception occur. This third section therefore provides an overview of the structure of how the NIC register write request handlers operate. The fourth section illustrates what kind of challenges that are encountered when Lemma V is to be proved for the NIC register write request handlers. This is done by describing how Lemma V can be proved for the most complex handler *cppi_ram_handler* by considering a computation path that contains a special case. The last section motivates why the oracle in the ideal model is atomic, which is a justified question considering what was presented in the fourth section.

B.1 Formal Definition of Ideal State

This section contains the formal definition of the ideal state. Since it is just an extension of the real state, by including the *oracle* variables, are the *cpu*, *memory* and *nic* components defined as for the real state. The comments describe how the oracle state shall be initialized. The definition follows the pseudocode notation in Appendix A and is as follows.

```
ideal_state = (
        cpu_state: cpu,
        word32 → word8: memory,
        nic_state: nic,
        oracle_state: oracle
)

oracle_state = (
        bool: kernel_running                //Initialized to true. True if the Linux kernel is running (not Linux application).
        bool: initialized,                  //Initialized to true. True if initialization process of the NIC has completed and is idle.
        bool: tx0_hdp_initialized,          //Initialized to true. True if the NIC has been reset and TX0_HDP is zeroed.
        bool: rx0_hdp_initialized,          //Initialized to true. True if the NIC has been reset and RX0_HDP is zeroed.
        bool: tx0_cp_initialized,           //Initialized to true. True if the NIC has been reset and TX0_CP is zeroed.
        bool: rx0_cp_initialized,           //Initialized to true. True if the NIC has been reset and RX0_CP is zeroed.
        bool: tx0_tearingdown,              //Initialized to false. If false, then is the transmission teardown process idle.
        bool: rx0_tearingdown,              //Initialized to false. If false, then is the reception teardown process idle.
        word32: tx0_active_queue, //Initialized to zero. All transmission descriptors in use by the NIC is in this queue.
        word32: rx0_active_queue, //Initialized to zero. All reception descriptors in use by the NIC is in this queue.
        word11 → bool: α,                   //Initialized to zero. α(w) is true if the wth word of CPPI_RAM is used by the NIC.
        word11 → word32: recv_bd_nr_blocks,         //Initialized to zero. Number of blocks that can be written by a descriptor.
        word20 → word32: ρNIC,              //Initialized to zero. Number of descriptors that can write the block bl.
        word20 → {L1, L2, D, MN, N, ⊥}: τ, //Initial value depends on initial memory mapping. Maps a block index to type.
        word20 → word32: ρex,               //Initialized according to initial memory mapping. Number of ex entries of a block.
        word20 → word32: ρwt,               //Initialized according to initial memory mapping. Number of wt entries of a block.
        {wordx}: GI,                                //Set of signatures that represent secure block content.
        word32768 → wordx: sign  //The signature function that is used to compute the signature of the content of a block.
)
```

B.2 Memory Mapping Request Handlers

The following subsections presents and describes the requirements, including the NIC extensions, of the memory mapping request handlers for accepting memory mapping requests issued by Linux, and how these handlers update the ideal state. It is upon these formal requirements and state updates that it has been reasoned that the memory mapping request handlers preserve *S*. Also worth to mention is that *mapL1*/*mapL2* and *createL1*/*createL2* might communicate with the NIC to update ρ_{NIC} to get an accurate view of which blocks the NIC might write, in order to not reject requests that are actually valid.

B.2.1 Switch

switch(*i*, *bl*) makes the MMU start its translation table walks from the block *bl* by setting *TTBR0* to point to *bl*.

The requirements are that *bl* is of type *L1* and that *bl* maps the code of the hypervisor and the monitor as expected:

$i.oracle.\tau(bl) = L1$ ∧
[$\forall s$ ∈ ideal_state.
    $s.memory = i.memory$ ∧ $s.cpu.cp15.DACR = i.cpu.cp15.DACR$ ∧
    $s.cpu.cp15.TTBR0[31{:}12] = bl$
    ⇒
    $HVM\_MAP(s)$].

HVM_MAP is defined in Subsection D.3.4. The state *s* is equal to *i* with respect to the state components that are relevant for the *mmu* function except for *TTBR0*. *TTBR0* is set to the new block *bl* in the state *s* to check that the MMU performs the hypervisor and monitor memory mappings as expected after *TTBR0* has been set to *bl* in the state *i*.

switch is defined as:

```
(ideal_state, bool): switch(ideal_state: i, word20: bl):
    i.cpu.cp15.TTBR0[31:12] := bl.
```

B.2.2 freeL1 and freeL2

freeL1(*i*, *bl*) and *freeL2*(*i*, *bl*) changes the type of *bl* to *D*. The requirements of *freeL1* and *freeL2* are:

- For *freeL1*: *bl* is of type *L1* and not pointed to by TTBR0:

  $i.oracle.\tau(bl) = L1$ ∧ $i.cpu.cp15.TTBR0[31{:}12] \neq bl$.

- For *freeL2*: *bl* is of type *L2* and not linked by an *L1* block:

  $i.oracle.\tau(bl) = L2$ ∧
  [$\neg\exists pt$ ∈ word20.
      $i.oracle.\tau(pt) = L1$ ∧ [$\exists pte \in L2\_LINK$(i, pt). $pte[31{:}12] = bl$]].

L2_LINK(*i*, *pt*) returns the list of all second-level link entries in the first-level page table block *pt*. *freeL1* and *freeL2* are defined as follows:

```
(ideal_state, bool): freeL1(ideal_state: i, word20: bl):
    i.oracle.τ(bl) := D
    i := decrement(i, PTL1(i, bl), length(PTL1(i, bl)))
    return i
```

```
(ideal_state, bool): $freeL2$(ideal_state: $i$, $word20$: bl):
    $i.oracle.\tau(bl) := D$
    $i := decrement(i, PTL2(i, bl), length(PTL2(i, bl)))$
    return $i$.
```

The type of the block *bl* is set to *D*, and for each block that *bl* maps as executable or writable is the reference counter ρ_{ex} and ρ_{wt} decremented by one, respectively. *PTL1* and *PTL2* are defined in Section D.1 (return the list of which access rights each block that is mapped by *bl* has), and *decrement* is defined as follows:

```
ideal_state: $decrement$(ideal_state: $i$, [(word20, bool, bool, bool)] $l$, nat: $j$):
    if $j = 0$ then
        return $i$
    else
        (word20: $pb$, bool: $rd$, bool: $wt$, bool: $ex$) := $l[j-1]$
        if $ex$ then $i.oracle.\rho_{ex}(pb)$ :-= 1
        if $wt$ then $i.oracle.\rho_{wt}(pb)$ :-= 1
        return $decrement(i, l, j-1)$.
```

B.2.3 unmapL1 and unmapL2

unmapL1(*i*, *bl*, *e*) and *unmapL2*(*i*, *bl*, *e*) free the entry with index *e* in the *L1*/*L2* block with index *bl*.(There are 1024 entries of four bytes each in a 4 kB block, indexed from 0 to 1023, inclusive.) The requirements of *unmapL1* and *unmapL2* are:

- The block *bl* is of type *L1*/*L2* and is not executable:

  $i.oracle.\tau(bl) = L1/L2 \wedge i.oracle.\rho_{ex}(bl) = 0$.

  This requirement prevents the oracle from inserting unsigned code into potential page tables that are executable.

- If *bl* is currently being used by the MMU, then the freed entry *e* must not change the hypervisor and monitor memory mappings. That is, virtual addresses belonging to the hypervisor or the monitor are still mapped to their expected physical addresses:

  $$\begin{array}{l}
  \forall s \in \text{ideal\_state}. \\
  \quad [\forall bl' \in \text{word20}.\ bl' \neq bl \Rightarrow content(s, bl') = content(i, bl')] \wedge \\
  \quad [\forall 0 \leq j < 4096. \\
  \quad\quad j \neq 4 \cdot e \Rightarrow s.memory(bl :: 0^{12} + j) = i.memory(bl :: 0^{12} + j)] \wedge \\
  \quad s.memory(bl :: 0^{12} + 4 \cdot e)\ \&\ 0b11 = 0b00 \wedge \\
  \quad s.cpu.cp15.TTBR0[31:12] = i.cpu.cp15.TTBR0[31:12] \wedge \\
  \quad s.cpu.cp15.DACR = i.cpu.cp15.DACR \\
  \quad \Rightarrow \\
  \quad HVM\_MAP(s).
  \end{array}$$

  The only difference between *s* and *i*, with respect to the operation of the MMU, is the entry *e* of the block *bl*, which is free in the state *s*, as stated by the third conjunct.

unmapL1 and *unmapL2* free the entry *e* of the block *bl* and decrement the reference counters ρ_{wt} and ρ_{ex} by one for each block that gets unmapped:

```
(ideal_state, bool): $unmapL1$(ideal_state: $i$, word20: $bl$, word10: $e$):
    $i.memory(\mathrm{bl} :: 0^{12} + 4\cdot e) := 0$
    $i := decrement(i, PTEL1(i, bl, e), length(PTEL1(i, bl, e)))$
    return $i$
```

```
(ideal_state, bool): $unmapL2$(ideal_state: $i$, word20: $bl$, word10: $e$):
    $i.memory(bl :: 0^{12} + 4\cdot e) := 0$
    $i := decrement(i, PTEL2(i, bl, e), length(PTEL2(i, bl, e)))$
    return $i$,
```

where *PTEL1*/*PTEL2*(*i*, *bl*, *e*) returns the list of blocks with their access rights that are mapped by the entry *e* in the page table block *bl* in the state *i*:

[(word20, bool, bool, bool)]: *PTEL1*/*PTEL2*(ideal_state: *i*, word20: *bl*, word32: *e*).

PTEL1 returns 256 entries since first-level entries map 1 MB of memory, or 256 consecutive 4 kB memory blocks, while *PTEL2* returns only one entry since second-level entries map 4 kB memory blocks. *PTEL1* and *PTEL2* return the empty list if the entry is already free.

B.2.4 linkL1

linkL1(ideal_state: *i*, word20: *bl*, word10: *e*, word20: *bl'*) sets entry *e* of the *L1* block *bl* to point to the second-level page table in the *L2* block *bl'*. The requirements of *linkL1* are:

- bl is of type L1 and not executable, and bl' of type L2:

  $i.oracle.\rho_{ex}(bl) = 0 \wedge i.oracle.\tau(bl) = L1 \wedge i.oracle.\tau(bl') = L2.$

- Hypervisor and monitor memory is mapped as expected:

$$\begin{aligned}
&\forall s \in \text{ideal\_state}.\\
&\quad [\forall bl'' \in \text{word20}.\ bl'' \neq bl \Rightarrow content(s, bl'') = content(i, bl'')] \wedge\\
&\quad [\forall 0 \leq j < 4096.\\
&\qquad j \notin [4\cdot e, 4\cdot e + 3] \Rightarrow s.memory(bl :: 0^{12} + j) = i.memory(bl :: 0^{12} + j)]\\
&\quad \wedge\\
&\quad s.memory(bl :: 0^{12} + 4\cdot e + 3) = bl'[19:12] \wedge\\
&\quad s.memory(bl :: 0^{12} + 4\cdot e + 2) = bl'[11:4] \wedge\\
&\quad s.memory(bl :: 0^{12} + 4\cdot e + 1)[7:2] = bl'[3:0] :: 0^{2} \wedge\\
&\quad s.memory(bl :: 0^{12} + 4\cdot e)[1:0] = 0b01 \wedge\\
&\quad s.cpu.cp15.TTBR0[31:12] = i.cpu.cp15.TTBR0[31:12] \wedge\\
&\quad s.cpu.cp15.DACR = i.cpu.cp15.DACR\\
&\quad \Rightarrow\\
&\quad HVM\_MAP(s).
\end{aligned}$$

This formula requires that all state components of *s* and *i* that affect the operation of the MMU to be equal, except for the entry *e* of the block *bl*, which is set to point to the second-level page table block *bl'*. Since *bl'* is a 4 kB block, and second-level entries can point to page tables of 1 kB, are the two least significant bits of the second-level entry zeroed in the fifth conjunct. Bit one and zero of a page table entry must be set to zero and one, respectively, to inform the hardware that the entry is a second-level link.

linkL1 is defined as:

```
(ideal_state, bool): $linkL1$(ideal_state: $i$, word20: $bl$, word10: $e$, word20: $bl'$):
  if $i.memory(bl :: 0^{12} + 4\cdot e)[1] = 1$ then
    $i := decrement(i, PTEL1(i, bl, e), length(PTEL1(i, bl, e)))$
  $i.memory(bl :: 0^{12} + 4\cdot e + 3) := bl'[19:12]$
  $i.memory(bl :: 0^{12} + 4\cdot e + 2) := bl'[11:4]$
  $i.memory(bl :: 0^{12} + 4\cdot e + 1)[7:2] := bl'[3:0] :: 0^2$
  $i.memory(bl :: 0^{12} + 4\cdot e)[1:0] :=$ 0b01
  return $i$.
```

If the entry *e* is already mapping blocks, and that mapping gives executable or writable access, then the mapped blocks gets their entries in ρ_{ex} and ρ_{wt} decremented by one, and entry *e* of the *L1* block *bl* is set to point to the second-level page table in the *L2* block *bl'*. Bits eight to five must also be set to specify the domain. That value depends on whether the memory that is mapped by the block *bl'* belongs to the hypervisor, the monitor or Linux. That is left unspecified.

B.2.5 mapL1 and mapL2

mapL1(*i*, *bl*, *e*, *bl'*, *rd*, *wt*, *ex*) and *mapL2*(*i*, *bl*, *e*, *bl'*, *rd*, *wt*, *ex*) maps entry *e* of the *L1*/*L2* block *bl* to the block *bl'* with the access permissions as specified by the arguments *rd*, *wt* and *ex*. The requirements are:

- The block *bl* is of type *L1*/*L2* and is non-executable:

  $i.oracle.\tau(bl) = L1/L2 \land i.oracle.\rho_{ex}(bl) = 0.$

- All mapped blocks *bl''* (first-level page tables map 256 consecutive blocks starting from *bl'*, where *bl''* is referred to as *bl'* + *j* below) that refer to hypervisor or monitor memory are inaccessible:

  - For *mapL1*:

    $\forall j \in \text{word8}.\ bl' + j \in HYP\_BL \cup MON\_BL \Rightarrow \neg rd \land \neg wt \land \neg ex.$

  - For *mapL2*:

    $bl' \in HYP\_BL \cup MON\_BL \Rightarrow \neg rd \land \neg wt \land \neg ex.$

  HYP_BL contains all block indexes that are allocated to the hypervisor, including code an data, and similarly for *MON_BL* for the monitor.

- If executable access permission is requested, then all mapped blocks *bl''* contain signed code:

  - For *mapL1*:

    $\forall j \in \text{word8}.\ ex \Rightarrow i.oracle.sign(content(i, bl' + j)) \in i.oracle.GI.$

  - For *mapL2*:

    $ex \Rightarrow i.oracle.sign(content(i, bl')) \in i.oracle.GI.$

- All mapped blocks *bl''* must belong to valid memory (hypervisor, monitor or Linux memory or NIC registers), the requested access permissions must not be both executable and writable or be in conflict with any other page table entry of any *L1*/*L2* block (including *bl*), and if writable access permission is requested then all mapped blocks must be of type *D*:
  - For *mapL1*:

    $$\begin{aligned}&\forall j \in \text{word8}.\\&\quad bl' + j \in HYP\_BL \cup MON\_BL \cup LINUX\_BL \cup NIC\_BL \land\\&\quad (ex \Rightarrow \neg wt \land i.oracle.\rho_{wt}(bl' + j) = 0) \land\\&\quad (wt \Rightarrow \neg ex \land i.oracle.\rho_{ex}(bl' + j) = 0 \land i.oracle.\tau(bl' + j) = D).\end{aligned}$$

  - For *mapL2*:

    $$\begin{aligned}&bl' \in HYP\_BL \cup MON\_BL \cup LINUX\_BL \cup NIC\_BL \land\\&(ex \Rightarrow \neg wt \land i.oracle.\rho_{wt}(bl') = 0) \land\\&(wt \Rightarrow \neg ex \land i.oracle.\rho_{ex}(bl') = 0 \land i.oracle.\tau(bl') = D).\end{aligned}$$

  Where *LINUX_BL* contains the block indexes of all blocks that are allocated to Linux, and *NIC_BL* = {0x4A100, 0x4A101, 0x4A102, 0x4A103} contains the block indexes of the NIC registers. This requirement prevents Linux from writing unsigned code into executable blocks, potential page tables or outside Linux RAM.
- If the *L1*/*L2* block *bl* is currently being used by the MMU, then the memory mappings of the hypervisor and the monitor must be as expected:
  - For *mapL1*:

    $$\begin{aligned}&\forall s \in \text{ideal\_state}.\\&\quad [\forall bl'' \in \text{word20}.\ bl'' \neq bl \Rightarrow content(s, bl'') = content(i, bl'')] \land\\&\quad [\forall 0 \leq j < 4096.\\&\qquad j \notin [4 \cdot e, 4 \cdot e + 3]\\&\qquad \Rightarrow\\&\qquad s.memory(bl :: 0^{12} + j) = i.memory(bl :: 0^{12} + j)] \land\\&\quad s.memory(bl :: 0^{12} + 4 \cdot e + 3) = bl'[19{:}12] \land\\&\quad s.memory(bl :: 0^{12} + 4 \cdot e + 2)[7{:}4] = bl'[11{:}8] \land\\&\quad s.memory(bl :: 0^{12} + 4 \cdot e)[1] = 0b1 \land\\&\quad s.cpu.cp15.TTBR0[31{:}12] = i.cpu.cp15.TTBR0[31{:}12] \land\\&\quad s.cpu.cp15.DACR = i.cpu.cp15.DACR\\&\quad \Rightarrow\\&\quad HVM\_MAP(s).\end{aligned}$$

    The 20 least significant bits of the entry *e* of block *bl* in this formula (except for bit one to specify that 1 MB block of memory is mapped, and because of that are only the 12 most significant bits needed) are unspecified for simplicity but should specify the access permissions according to *rd*, *wt* and *ex*.
  - For *mapL2*: Similar as for *mapL2* but with the 20 most significant bits of the entry set to *bl'*:

$s.memory(bl :: 0^{12} + 4\cdot e + 3) = bl'[19:12]\ \wedge$
$s.memory(bl :: 0^{12} + 4\cdot e + 2) = bl'[11:4]\ \wedge$
$s.memory(bl :: 0^{12} + 4\cdot e + 1)[7:4] = bl'[3:0].$

Requirements related to the NIC:

- If the mapped blocks *bl''* correspond to NIC registers, then must the access permissions be non-executable, and if they correspond to registers that affect which memory accesses the NIC does, then must the access permissions be non-writable:
  - ○ For *mapL1*:

    $[\forall j \in \text{word8}.$
    $\quad bl' + j \in \{\text{0x4A100, 0x4A101, 0x4A102, 0x4A103}\} \Rightarrow \neg ex]\ \wedge$
    $[\forall j \in \text{word8}.\ bl' + j \in \{\text{0x4A100, 0x4A102, 0x4A103}\} \Rightarrow \neg wt].$

  - ○ For *mapL2*:

    $bl' \in \{\text{0x4A100, 0x4A101, 0x4A102, 0x4A103}\} \Rightarrow \neg ex]\ \wedge$
    $bl' \in \{\text{0x4A100, 0x4A102, 0x4A103}\} \Rightarrow \neg wt].$

  This requirement prevents Linux from executing unsigned code, by interpreting NIC registers as instructions that can be modified arbitrarily by the NIC. Also, Linux cannot configure the NIC to enter an insecure state.

- If executable permission is requested then the blocks *bl''* must not be accessed by any receive buffer descriptor in the queue pointed to by *i.oracle.rx0_active_queue*:
  - ○ For *mapL1*:

    $\forall j \in \text{word8}.\ ex \Rightarrow i.oracle.\rho_{NIC}(bl' + j) = 0.$

  - ○ For mapL2:

    $ex \Rightarrow i.oracle.\rho_{NIC}(bl') = 0.$

  This prevents the NIC from writing unsigned code in executable blocks.

The operations performed by *mapL1* are:

```
(ideal_state, bool): mapL1(ideal_state: i, word20: bl, word10: e, word20: bl',
                                        bool: rd, bool: wt, bool: ex):
    if $i.memory(bl :: 0^{12} + 4\cdot e)[1] = 1$ then
        $i := decrement(i, PTEL1(i, bl, e), length(PTEL1(i, bl, e)))$
    $i.memory(bl :: 0^{12} + 4\cdot e + 3) := bl'[19:12]$
    $i.memory(bl :: 0^{12} + 4\cdot e + 2)[7:4] := bl'[11:8]$
    $i.memory(bl :: 0^{12} + 4\cdot e)[1] := 0b1$
    $i := increment(i, PTEL1(i, bl, e), length(PTEL1(i, bl, e)))$
    $i := set\_type\_D(i, bl', 256)$
    return $i$,
```

where *increment* increments the reference counters ρ_{wt} and ρ_{ex} for all blocks that are now mapped as writable and executable, respectively:

```
ideal_state: $increment$(ideal_state: $i$, [(word20, bool, bool, bool)] $l$, nat: $j$):
    if $j = 0$ then
        return $i$
    else
        (word20: $pb$, bool: $rd$, bool: $wt$, bool: $ex$) := $l[j - 1]$
        if $ex$ then $i.oracle.\rho_{ex}(pb)$ :+= 1
        if $wt$ then $i.oracle.\rho_{wt}(pb)$ :+= 1
        return $increment(i, l, j - 1)$,
```

and *set_type_D* sets all blocks that are currently typed as $\perp$ as *D* blocks, except for hypervisor and monitor blocks:

```
ideal_state: $set\_type\_D$(ideal_state: $i$, word20: $bl'$, nat: $j$):
    if $j = 0$ then
        return $i$
    else if $\neg(bl' + j - 1 \in HYP\_BL \cup MON\_BL) \wedge i.oracle.\tau(bl' + j - 1) = \perp$
        $i.oracle.\tau(bl' + j - 1) := D$
        return $set\_type\_D(i, bl', j - 1)$.
```

Again, the setting of access permissions and other management bits are omitted for simplicity. The operations of *mapL2* are:

```
(ideal_state, bool): $mapL2$(ideal_state: $i$, word20: $bl$, word10: $e$, word20: $bl'$,
                                                   bool: $rd$, bool: $wt$, bool: $ex$):
    if $i.memory(bl :: 0^{12} + 4 \cdot e)[1] = 1$ then
        $i := decrement(i, PTEL2(i, bl, e), length(PTEL2(i, bl, e)))$
    $i.memory(bl :: 0^{12} + 4 \cdot e + 3) := bl'[19:12]$
    $i.memory(bl :: 0^{12} + 4 \cdot e + 2) := bl'[11:4]$
    $i.memory(bl :: 0^{12} + 4 \cdot e + 1)[7:4] := bl'[3:0]$
    $i.memory(bl :: 0^{12} + 4 \cdot e + 0)[1] :=$ 0b1
    $i := increment$(i, $PTEL2(i, bl, e), length(PTEL2(i, bl, e)))$
    $i := set\_type\_D(i, bl', 1)$
    return $i$.
```

B.2.6 createL1 and createL2

createL1(*i*, *bl*) and *createL2*(*i*, *bl*) makes the block bl of type *L1* or *L2*, respectively, by validating their page table entries. Their requirements are:

- The block *bl* is of type *D* and is not writable to prevent Linux from changing access rights:

  $i.oracle.\tau(bl) = D \wedge i.oracle.\rho_{wt}(bl) = 0$.

  Requiring that *bl* is of type *D* has no practical limitations since making an *L1* block an *L1* block makes no difference, and switching types of *L1* and *L2* blocks makes no sense since *L1* and *L2* blocks have different page table entry formats.

- All blocks that map to hypervisor or monitor memory are mapped as inaccessible:

$\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).$
  $bl' \in HYP\_BL \cup MON\_BL \Rightarrow \neg rd \wedge \neg wt \wedge \neg ex.$

- All second-level entries must link to *L2* blocks:

  $L2\_ENTRY\_L2\_BL(i, bl).$

  L2_ENTRY_L2_BL is defined in Subsection D.3.2.

- For each mapped block *bl'* that is requested to be executable, it must contain signed code:

  $\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).$
    $ex \Rightarrow i.oracle.sign(content(i, bl')) \in i.oracle.GI.$

- For each mapped block *bl'*, it must belong to a valid memory region and not be both executable and writable or be in conflict with any other page table entry of any other *L1*/*L2* block, and if writable access permission is requested then it must be of type *D*:

  $\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).$
    $bl' \in HYP\_BL \cup MON\_BL \cup LINUX\_BL \cup NIC\_BL \wedge$
    $(ex \Rightarrow \neg wt \wedge \text{i}.oracle.\rho_{wt}(bl') = 0) \wedge$
    $(wt \Rightarrow \neg ex \wedge i.oracle.\rho_{ex}(bl') = 0 \wedge i.oracle.\tau(bl') = D).$

- No pair of entries in *bl* are in conflict with each other with respect to the write and execute permissions:

  $\forall(pb, rd, wt, ex) \in PTL1/PTL2(i, bl).$
    $\neg\exists(pb', rd', wt', ex') \in PTL1/PTL2(i, bl).$
      $pb' = pb \wedge (ex \wedge wt' \vee ex' \wedge wt).$

- If the block *bl* maps itself, then it must not be writable:

  $\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).\ bl' = bl \Rightarrow \neg wt.$

Requirements related to the NIC:

- The new *L1*/*L2* block *bl* cannot be referred to by any receive buffer descriptor or correspond to NIC registers:

  $i.oracle.\rho_{NIC}(bl) = 0 \wedge bl \notin NIC\_BL.$

  This prevents the NIC from changing access permissions in potential page tables.

- No entry in the *L1*/*L2* block *bl* can map a block *bl'* that corresponds to the NIC registers with write or execute access permission:

  $\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).\ bl' \in NIC\_BL \Rightarrow \neg wt \wedge \neg ex.$

- All blocks *bl'* that are mapped as executable cannot be referred to by receive buffer descriptors:

  $\forall(bl', rd, wt, ex) \in PTL1/PTL2(i, bl).\ ex \Rightarrow i.oracle.\rho_{NIC}(bl') = 0.$

The operations performed by *createL1* are:

```
(ideal_state, bool): createL1(ideal_state: i, word20: bl):
    i.oracle.τ(bl) := L1
    i := increment(i, PTL1(i, bl), length(PTL1(i, bl)))
    i := set_type_D_for_list(i, PTL1(i, bl), length(PTL1(i, bl)))
    return i,
```

where *set_type_D_for_list* sets all blocks that are currently typed as ⊥ as *D* blocks, except for hypervisor and monitor blocks:

```
ideal_state: set_type_D_for_list(ideal_state: i, [(word20, bool, bool, bool)]: l,
                                                                          nat: j):
    if j = 0 then
        return i
    else
        (word32: bl, bool: rd, bool: wt, bool: ex) := l[j – 1]
        if i.oracle.τ(bl) = ⊥ ∧
            ¬(bl ∈ HYP_BL ∪ MON_BL) then i.oracle.τ(bl) := D
        return set_type_D_for_list(i, l, j – 1).
```

```
(ideal_state, bool): createL2(ideal_state: i, word20: bl):
    i.oracle.τ(bl) := L2
    i := increment(i, PTL2(i, bl), length(PTL2(i, bl)))
    i := set_type_D_for_list(i, PTL2(i, bl), length(PTL2(i, bl)))
    return i.
```

B.3 NIC Register Write Request Handlers

This section includes pseudocode for the representative NIC Register write request handlers and shows their fundamental operation and how they are integrated into the data abort exception handler. Their auxiliary functions that deal with the deeper details of buffer descriptors are omitted to save space. It is these formal definitions that have been used to reason that the NIC register write request handlers preserve the formal definition of *S* in appendix D. The pseudocode follows.

```
//Addresses of NIC DMA controller registers.
word32: CPSW_CPDMA := 0x4A10_0800
word32: TX_TEARDOWN := CPSW_CPDMA + 0x8
word32: RX_TEARDOWN := CPSW_CPDMA + 0x18
word32: CPDMA_SOFT_RESET := CPSW_CPDMA + 0x1C
word32: DMACONTROL := CPSW_CPDMA + 0x20
word32: RX_BUFFER_OFFSET := CPSW_CPDMA + 0x28

//Addresses of NIC HDP and CP registers.
word32: CPSW_STATERAM := 0x4A10_0A00
word32: CPSW_STATERAM_SIZE := 0x80
word32: TX0_HDP := CPSW_STATERAM + 0x0
word32: RX0_HDP := CPSW_STATERAM + 0x20
word32: TX0_CP := CPSW_STATERAM + 0x40
word32: RX0_CP := CPSW_STATERAM + 0x60

//Address range constants of CPPI_RAM.
word32: CPPI_RAM := 0x4A10_2000
word32: CPPI_RAM_SIZE := 0x2000

//Constants related to updating transmission and reception queues.
bool: TRANSMIT := true    //Operation is with respect to the transmission queue.
bool: RECEIVE := false    //Operation is with respect to the reception queue.
bool: ADD := true         //Added buffer descriptors to queue.
bool: REMOVE := false     //Removed buffer descriptors from queue.
```

```
//Teardown interrupt code.
word32: TD_INT := 0xFFFF_FFFC

//Constants related to CPPI_RAM writes.
word32: ZEROED_NDP_OVERLAP := 0    //Writing last buffer descriptor's next descriptor pointer .
word32: ILLEGAL_OVERLAP := 1       //Writing a buffer descriptor word that is not a zeroed next descriptor pointer.
word32: NO_OVERLAP := 2                 //Writing an unused word of CPPI_RAM.

/*
 *     data_abort is the function that the ideal CPU applies when a data abort
 *     exception has occurred. When that occurs, the CPU is in abt mode and the
 *     program counter is set to 0xFFFF0010.
 *
 * If the accessed virtual address that generated the exception is mapped to
 *     the registers of the NIC and the Linux kernel is running, then the NIC
 *     register write request handlers checks the write request and executes it
 *     only if it is secure. If it is not secure or some other error is
 *     encountered, then is NIC register write request denied and either does
 *     Linux continue its execution from the instruction following the one that
 *     failed, or does a memory mapping request handler handle the exception.
 *
 *     If the data abort exception was due to some other reason, then the
 *     unspecified handler handle_data_abort_exception handles that data abort
 *     exception. The handler handle_data_abort_exception is left
 *     unspecified since it is not directly security critical and just forwards
 *     the data abort exception to the data abort exception handler of the Linux
 *     kernel.
 *
 *     This function returns the new ideal system state.
 */
ideal_state: data_abort(ideal_state: i)

    if i.oracle.kernel_running then
        //Retrieves the physical address that Linux tried to write. If it
        //belongs to a NIC register, then is check_nic_access applied to handle
        //the NIC register write request. Otherwise is the data abort treated
        //as a data abort exception that shall be forwarded to Linux or to a
        //memory mapping request handler.
        word32: pa := mmu(i, PL1, i.cpu.cp15.DFAR, rd)

        if valid ∧ 0x4A10_0000 ≤ pa ∧ pa < 0x4A10_4000 then
            //The flag accepted indicating whether the NIC register write
            //request was accepted or denied is unused for the moment but can
            //be used to tell if Linux is doing something strange (being under
            //attack).
            (i, bool: accepted) := check_nic_access(i, pa)

            //Restores the program counter to the instruction following the one
            //that issued the NIC register write request.
            i.cpu.uregs.r15 := i.cpu.pregs.r14_abt - 0x4

            //Restores CPSR to its value immediately before the exception
            //occurred. The mask 0xFFFFFF30 ensures that Linux executes in user
            //mode and that IRQ and FIQ interrupts are enabled.
            i.cpu.sregs.CPSR := i.cpu.sregs.SPSR_abt & 0xFFFFFF30

            //Returns the new state where Linux is ready to execute again.
            return i

    //The Linux kernel was not running or a NIC register write request was not
    //made.
    return handle_data_abort_exception(i)

/*
 *     Applies the NIC register write request handler for the NIC register that
 *     Linux tried to write at physical address pa_nic_register. This is done by
 *     first retrieving the content of the CPU register that was used by the
 *     instruction that tried to perform the store into the NIC register. Then is
 *     the handler that handles accesses to that NIC register applied. This
 *     function returns the updated ideal system state that is the result of
 *     applying the NIC register write request handler and a flag that tells
 *    whether the NIC register write request was accepted or not (true means
 *    executed and false means rejected).
 */
(ideal_state, bool): check_nic_access(ideal_state: i, word32: pa_nic_register)
    //Virtual address of instruction that raised the data abort exception.
    word32: va_instruction := i.cpu.pregs.r14_abt - 0x8
    //If the address of the instruction or the NIC register are not 32-bit word
    //aligned, nothing is done.
    if va_instruction[1:0] ≠ 0 ∨ pa_nic_register[1:0] ≠ 0 then
        return (i, false)

    //Retrieves the encoding of the instruction that tried to write a NIC
    //register.
    word32: instruction_code := mmu(i, PL1, va_instruction, rd)

    //If no access is permitted to the failing instruction, nothing is done.
    if ¬valid then
        return (i, false)

    //If the instruction was not an ordinary store instruction, then nothing is
    //done. An ordinary store instruction have the following syntax:
    //STR    Rt, [Rn, #+imm32] which performs the following operation:
    //mem32[Regs[Rn] + imm32] := Regs[Rt]
    if 0xFFF00000 & instruction_code ≠ 0xE5800000 then
        return (i, false)

    //Computes the register index of the register that contains the value that
    //Linux wants to write to the NIC register at physical address
    //pa_nic_register.
    word32: t =: (0x0000F000 & instruction_code) >> 12

    //Retrieves the register content that Linux used when trying to write a NIC
    //register. value is Rt above, and va is Rn added with imm. Linux tried to
    //write value to physical address pa: memory(pa) := value.
    word32: value := user_register_content(i, t)

    //Applies the handler for the specifically accessed NIC register.
    if pa_nic_register = TX_TEARDOWN then
        return tx_teardown_handler(i, value)
    else if pa_nic_register = RX_TEARDOWN then
        return rx_teardown_handler(i, value)
    else if pa_nic_register = CPDMA_SOFT_RESET then
        return cpdma_soft_reset_handler(i, value)
    else if pa_nic_register = DMACONTROL then
        return dmacontrol_handler(i, value)
    else if pa_nic_register = RX_BUFFER_OFFSET then
        return rx_buffer_offset_handler(i, value)
    else if pa_nic_register = TX0_HDP then
        return tx0_hdp_handler(i, value)
    else if pa_nic_register = RX0_HDP then
        return rx0_hdp_handler(i, value)
    else if pa_nic_register = TX0_CP then
        return tx0_cp_handler(i, value)
    else if pa_nic_register = RX0_CP then
        return rx0_cp_handler(i, value)
    else if CPPI_RAM ≤ pa_nic_register ∧ pa_nic_register < CPPI_RAM + CPPI_RAM_SIZE then
        return cppi_ram_handler(i, pa_nic_register, value)
    else if CPSW_STATERAM ≤ pa_nic_register ∧
            pa_nic_register < CPSW_STATERAM + CPSW_STATERAM_SIZE then
        return stateram_handler(i, value)
    else
        return write_nic_register_handler(i, pa_nic_register, value)

/*
 *    Linux wants to write TX_TEARDOWN with the value channel to teardown a NIC
 *    transmission DMA channel.
 *
 *    value contains the ID of the transmit channel (zero to seven, inclusive) to
 *    tear down. Since only transmit channel zero is allowed to be used, value
 *    must be equal zero. If the NIC has not been initialized or an earlier
 *    transmit teardown operation has not finished, then this operation is also
 *    denied. Otherwise tx0_tearingdown is set to true since a transmit teardown
 *     operation is to be initiated and the transmit teardown operation is
 *     initiated by actually writing channel zero's ID number to the TX_TEARDOWN
 *     register.
 */
(ideal_state, bool): tx_teardown_handler(ideal_state: i, word32: channel)
       if ¬i.oracle.initialized ∨ i.oracle.tx0_tearingdown ∨ (channel & 0x7) ≠ 0 then
              return (i, false)
       else
              i.oracle.tx0_tearingdown := true
              i.nic := write_nic_register(i.nic, TX_TEARDOWN, 0)   //channel = 0.
              return (i, true)

/*
 *     Linux wants to write the CPDMA_SOFT_RESET register with the value val.
 *     Probably to reset the NIC DMA hardware logic.
 *
 *     The main purpose of cpdma_soft_reset_handler, except from initiating the
 *     initialization process of the NIC DMA hardware, is to check that such a
 *     write does not put the NIC into a dead state. According to the NIC model
 *     definition of write_cpdma_soft_reset, the NIC enters a dead state when
 *     setting CPDMA_SOFT_RESET to one if any of the following conditions hold:
 *     -The NIC DMA hardware reset operation followed by the zeroing of the HDP
 *      and CP registers is not complete.
 *     -The transmit or receive teardown processes are active.
 *
 *     To prevent putting the NIC in a dead state, the two if statements are used.
 *     The first one checks if a reset operation is to be initiated. If Linux just
 *     wants to write zero, which has no effect, the handler does nothing. The
 *     second check ensures that the initialization and teardown processes are
 *     idle. This is done by using the oracle variables initialized and
 *     tx0/rx0_tearingdown which track the progress of those NIC processes.
 *
 *     Otherwise is the initialization process of the NIC initiated and the oracle
 *     variables that track the initialization process are initialized.
 */
(ideal_state, bool): cpdma_soft_reset_handler(ideal_state: i, word32: val)
       if (val & 0b1) = 0 then
              return (i, true)
       else if ¬i.oracle.initialized ∨ i.oracle.tx0_tearingdown ∨ i.oracle.rx0_tearingdown then
              return (i, false)
       else
              i.oracle.initialized := false
              i.oracle.tx0_hdp_initialized := false
              i.oracle.rx0_hdp_initialized := false
              i.oracle.tx0_cp_initialized := false
              i.oracle.rx0_cp_initialized := false
              i.nic := write_nic_register(i.nic, CPDMA_SOFT_RESET, 1)  //val = 1.
              return (i, true)

/*
 *     Linux wants to write TX0_HDP with the value bd_ptr which is the head of a
 *     buffer descriptor queue, in order to send the data buffers associated with
 *     the buffer descriptors in that queue.
 *
 *     The argument bd_ptr contains the physical address of the first buffer
 *     descriptor in a queue that Linux wants to transmit. If initialized is
 *     false, then this tells the oracle that the initialization process of the
 *     NIC is under progress. In that case TX0_HDP is supposed to be set to zero
 *     to make the initialization process progress, but only if the hardware reset
 *     operation is complete CPDMA_SOFT_RESET is zero).
 *
 *     If all HDP and CP registers have been initialized, then
 *     initialization_performed is called which updates ρNIC, α, and
 *     recv_bd_nr_blocks to zero for the buffer descriptors in the queue pointed
 *     to by tx0_active_queue since they are unused by the NIC after an
 *     initialization and therefore tx0_active_queue is also zeroed.
 *
 *     If initialized is true, then it is checked that TX0_HDP is zero (otherwise
 *     it is an error to write it) and that the NIC transmit teardown process is
 *     idle since it is undefined what happens if the transmission process is
 *     started during a transmit teardown process.
 *
 *      Then ρNIC, α, recv_bd_nr_blocks and tx0_active_queue are
 *      updated to reflect the fact that the transmission process is idle (in the
 *      sense that all buffer descriptors in the transmission queue are released
 *      since TX0_HDP is zero) and the released buffer descriptors are inactive.
 *      Before the new queue, pointed to by bd_ptr, is allowed to start
 *      transmission it must be checked to be secure.
 *
 *      If the new queue is secure, then tx0_active_queue is set to point to its
 *      head and ρNIC, α and recv_bd_nr_blocks are updated for the
 *      new transmit buffer descriptors. Finally TX0_HDP is written with the head
 *      of the new queue to actually start the transmission process.
 */
(ideal_state, bool): tx0_hdp_handler(ideal_state: i, word32: bd_ptr)
    if ¬i.oracle.initialized then                              //NIC is not initialized.
        (i.nic, word32: cpdma_soft_reset) := read_nic_register(i.nic, CPDMA_SOFT_RESET)
        if cpdma_soft_reset = 1 ∨ bd_ptr ≠ 0 then    //Reset not complete or incorrect initialization value.
            return (i, false)
        else
            i.nic := write_nic_register(i.nic, TX0_HDP, 0)       //bd_ptr = 0.
            i.oracle.tx0_hdp_initialized := true       //TX0_HDP initialized.

            //Checks if all HDP and CP registers are initialized. If so, the
            //tx0/rx0_active_queue queues are emptied and related data
            //structures initialized.
            if i.oracle.rx0_hdp_initialized ∧ i.oracle.tx0_cp_initialized ∧ i.oracle.rx0_cp_initialized then
                i := initialization_performed(i)

            return (i, true)
    else                                                                 //NIC is initialized.
        //If TX0_HDP is not zero or a teardown is under progress, then the NIC
        //register write request to TX0_HDP is denied.
        (i.nic, word32: tx0_hdp) := read_nic_register(i.nic, TX0_HDP)
        if tx0_hdp ≠ 0 ∨ i.oracle.tx0_tearingdown then
            return (i, false)
        //In this case TX0_HDP is zero and no transmission teardown is under
        //progress. tx0_active_queue and related data structures are updated to
        //reflect the current state of the NIC. That is the transmission queue
        //is empty. Then it is checked whether the new queue is secure and if
        //so, it is added to tx0_active_queue and related data structures are
        //updated. Finally TX0_HDP is set to the buffer descriptor at the head
        //of the new queue.
        else
            i := update_active_queue(i, TRANSMIT)
            (i, bool: is_queue_secure) := is_queue_secure(i, bd_ptr, TRANSMIT)
            if is_queue_secure then
                i.oracle.tx0_active_queue := bd_ptr
                i := update_ρNIC_α_queue(i, bd_ptr, TRANSMIT, ADD)
                i.nic := write_nic_register(i.nic, TX0_HDP, bd_ptr)
                return (i, true)
            else
                return (i, false)

/*
 *      Linux wants to write TX0_CP with the value val to acknowledge a
 *      frame transmission completion interrupt or a transmission teardown
 *      interrupt.
 *
 *      The argument value that Linux tried to write to TX0_CP is only written if
 *      any of the following cases hold:
 *      -The NIC is in the initialization process and has finished the reset
 *       operation, and TX0_CP is to be initialized to zero. These requirements
 *       avoids putting the NIC in a dead state.
 *      -The NIC is not executing the initialization process or the transmission
 *       teardown process and therefore can TX0_CP be written to any value.
 *      -A teardown process is complete and have terminated the transmission
 *       process, and Linux wants to acknowledge this (val = TD_INT). A
 *       transmission teardown process is considered complete by the oracle only if
 *       TX0_CP contains the teardown interrupt code TD_INT = 0xFFFFFFFC,
 *       TX0_HDP = 0, and the following two conditions hold:
 *              -If there are unused buffer descriptors left in the queue pointed to by
 *               tx0_active_queue after the teardown, then the first of those unused
 *               buffer descriptors have its teardown bit set.
 *              -If there are no unused buffer descriptors, then all of the buffer
 *               descriptors in the transmission queue have been released.
 *       These two conditions imply that an update of tx0_active_queue is equal will
 *       assign it the value zero. If there is a set teardown bit in the first unused buffer
 *       descriptor or all buffer descriptors in the transmission queue are released by
 *       the NIC, then update_active_queue sets tx0_active_queue to zero and updates
 *       α. If the teardown process is considered complete, then the teardown is
 *       acknowledged and the NIC deasserts the teardown interrupt and
 *       tx0_tearingdown is falsified since the teardown is complete. The purpose with
 *       the last if statement is to make sure that the transmission teardown process is
 *       complete. The condition of this if statement requires that the last step of the
 *       transmit teardown process is to clear TX0_HDP or set TX0_CP to
 *       0xFFFFFFFC. This is consistent with the NIC model.
 *
 *      If Linux did not try to do something strange, true is returned. The reason
 *      for forcing Linux to acknowledge the teardown is because an atomic oracle
 *      cannot initiate the teardown process and then wait until the teardown is
 *      complete: The NIC cannot execute while the oracle executes since the oracle
 *      is atomic! Since a well-behaving Linux guest does acknowledge teardown
 *      interrupts this is not a practical problem, provided that CP registers are
 *      set to 0xFFFFFFFC by the NIC as the last operation of a teardown process,
 *      which is unknown as discussed in 4.2.10.4 Design Issues.
 *
 *      Some handlers force Linux to acknowledge teardowns before they allow the
 *      NIC register write. Sometimes it makes the verification task easier but
 *      most often is it to prevent the NIC from entering a dead state.
 */
(ideal_state, bool): tx0_cp_handler(ideal_state: i, word32: val)
    if ¬i.oracle.initialized then
        (i.nic, word32: cpdma_soft_reset) := read_nic_register(i.nic, CPDMA_SOFT_RESET)
        if cpdma_soft_reset = 1 ∨ val ≠ 0 then
            return (i, false)
        else
            i.nic := write_nic_register(i.nic, TX0_CP, 0) //val = 0.
            i.oracle.tx0_cp_initialized := true

            if i.oracle.tx0_hdp_initialized ∧ i.oracle.rx0_hdp_initialized ∧ i.oracle.rx0_cp_initialized then
                i := initialization_performed(i)

            return (i, true)
    else
        //No teardown under progress and any value can be written.
        if ¬i.oracle.tx0_tearingdown then
            i.nic := write_nic_register(i.nic, TX0_CP, val)
            return (i, true)

        //Updates tx0_active_queue and related data structures and reads TX0_CP
        //and TX0_HDP.
        i := update_active_queue(i, TRANSMIT)
        (i.nic, word32: tx0_cp) := read_nic_register(i.nic, TX0_CP)
        (i.nic, word32: tx0_hdp) := read_nic_register(i.nic, TX0_HDP)
        //If the teardown is considered complete and Linux wants to acknowledge
        //it, then the teardown interrupt is acknowledged and the
        //tx0_tearingdown flags is cleared since the teardown process is now
        //considered complete.
        if i.oracle.tx0_tearingdown ∧ i.oracle.tx0_active_queue = 0 ∧
                tx0_cp = TD_INT ∧ tx0_hdp = 0 ∧ val = TD_INT then
            i.nic := write_nic_register(i.nic, TX0_CP, TD_INT)      //val = TD_INT
            i.oracle.tx0_tearingdown := false
            return (i, true)
        else
            return (i, false)
```

B.4 cppi_ram_handler and Proof of Lemma V

This section presents the pseudocode of *cppi_ram_handler*, and describes how Lemma V can be proved for *cppi_ram_handler* for a special case. That scenario constitutes the most complex proof of Lemma V for a NIC register write request handler.

B.4.1 cppi_ram_handler

```
/*
 *      Linux wants to write the CPPI_RAM word with physical address
 *      pa with the value val, in to either extend an active queue
 *      or to initialize a buffer descriptor that will later be inserted into a
 *      transmission or reception queue.
 *
 *      The first argument contains the physical address of a word in CPPI_RAM that
 *      Linux tried to write with the value in the second argument.
 *
 *      To ease the simulation proof a check is first made to make sure that the
 *      initialization and teardown processes are idle. Then the
 *      tx0/rx0_active_queue, α and ρNIC variables are updated to
 *      give an accurate view of which CPPI_RAM words are in use by the NIC and
 *      hence can affect the operation of the NIC.
 *
 *      If the accessed CPPI_RAM word is not used by the NIC then it is written and
 *      the handler returns.
 *
 *      Otherwise the handler checks what kind of overlap the CPPI_RAM write
 *      request made on the transmit queue, and if no overlap with the transmit
 *      queue was made, then the receive queue is checked as follows:
 *      -Next descriptor pointer of last buffer descriptor: In this case it is
 *       checked if the new queue to be appended is secure. A queue is considered
 *       secure if:
 *              1.      Its buffer descriptors are word aligned in CPPI_RAM.
 *              2.      It does not overlap the queues pointed to by tx0/rx0_active_queue.
 *              3.      It does not overlap itself.
 *              4.      Its buffer descriptors only access Linux RAM memory and have buffer
 *                      length fields greater than zero.
 *              5.      For transmission, its SOP and EOP bits match (each SOP is matched
 *                      by an EOP without an intermediate SOP) and the packet length field
 *                      in the SOP buffer descriptors is equal to the sum of the buffer
 *                      descriptors belonging to the same frame.
 *              6.      For reception, no executable pages or page tables are accessed.
 *
 *       If is_queue_secure determines the queue secure, then it also configures
 *       some bits of the buffer descriptors appropriately:
 *              -For transmission buffer descriptors:
 *                      -Sets the ownership bit in SOPs.
 *                      -Clears EOQ bit in EOPs.
 *                      -Celars TD bit in SOPs.
 *              -For all reception buffer descriptors:
 *                      -Clears Buffer Offset, SOP, EOP, EOQ, TD and CRC fields.
 *                      -Sets the ownership bit.
 *
 *       If the new queue is secure, then:
 *              1.      α is updated to mark the buffer descriptors of the
 *                      new queue as active (ρNIC is not updated for the transmit case).
 *              2.      The requested value is written to the requested CPPI_RAM word.
 *              3.      A potential misqueue condition is handled. The reason for the
 *                      misqueue handling is to allow the ideal Linux model to simulate the
 *                      real Linux model with an atomic oracle. It just restarts
 *                      transmission from the new queue if the transmission process was
 *                      finished after tx0_active_queue was updated.
 *      -Any other overlap: Are considered illegal even if they are not security
 *       related. This actually caused an incompatibility problem with the
 *       operation of the Linux NIC driver. By changing one line in the Linux NIC
 *       driver, this problem was solved.
 *      -no overlap: If no overlap was made on the transmit queue, then the
 *       operations performed in the previous two bullets are done but for the
 *       receive case. For the receive case ρNIC is updated if the receive queue is
 *       extended. A secure queue does not need to have matching SOP and EOP buffer
 *       descriptors (since they are set by the NIC), and the new queue must only
 *       access non-executable, non-NIC register data blocks. If the queue is
 *       determined secure by is_queue_secure then the the buffer descriptors are
 *       configured as follows: clears Buffer Offset, SOP, EOP, EOQ, TD and CRC
 *       bits, and sets the ownership bit, in all buffer descriptors.
 *
 *      All in all, this handler accepts a CPPI_RAM write request if any of the two
 *      cases apply:
```

```
 *     -The access does not overlap the transmit and receive queues.
 *     -Extends the transmit or receive queues if the queue to add is secure, and
 *      then appropriately updates α, ρNIC and configures the added
 *      buffer descriptor flag bits.
 */
(ideal_state, bool): cppi_ram_handler(ideal_state: i, word32: pa, word32: val)
    //Checks that the initialization and teardown processes are idle.
    if ¬i.oracle.initialized ∨ i.oracle.tx0_tearingdown ∨ i.oracle.rx0_tearingdown then
        return (i, false)

    //Updates the oracles view of which buffer descriptors are in use by the
    //NIC by updating tx0/rx0_active_queue and related data structures. That
    //view might affect the oracle's decision of rejecting or accepting the NIC
    //register write request to CPPI_RAM.
    i := update_active_queue(i, TRANSMIT)
    i := update_active_queue(i, RECEIVE)

    //Checks if the accessed CPPI_RAM is used by the NIC. If not, then the
    //write can be executed and the handler returns.
    if ¬i.oracle.α((pa – CPPI_RAM) >> 2) then
        i.nic := write_nic_register(i.nic, pa, val)
        return (i, true)

    //Since the previous if statement failed, it means that the accessed
    //CPPI_RAM word was used by the NIC, as considered by the oracle, and
    //therefore it must be checked which word of an active buffer descriptor
    //that was accessed. First is the transmit queue checked and then the
    //receive queue. They are handled separately.
    word32: transmit_overlap := type_of_cppi_ram_access_overlap(i, pa, i.oracle.tx0_active_queue)
    //Only the next descriptor pointer of the last buffer descriptor in a queue
    //is allowed to be written.
    if transmit_overlap = ZEROED_NDP_OVERLAP then
        //If such a write is requested then it is checked whether the appended
        //queue queue is secure.
        (i, bool: is_queue_secure) := is_queue_secure(i, val, TRANSMIT)
        //If the queue is considered secure, then relevant to the new queue are
        //updated and the queue is appended by actually performing the write to
        //CPPI_RAM.
        if is_queue_secure then
            i := update_ρNIC_α_queue(i, val, TRANSMIT, ADD)
            i.nic := write_nic_register(i.nic, pa_nic_register, value)
            //To allow this atomic oracle to simulate an implementation of this
            //handler, a potential misqueue condition is also checked: If the NIC
            //completed the processing of the transmission queue just after
            //update_active_queue was applied but before it was extended, then is
            //the HDP register set to the new queue to activate transmission for the
            //new buffer descriptors.
            i := handle_potential_misqueue_condition(i, TRANSMIT, pa, val)
            return (i, true)
        else
            return (i, false)
    //The overlap on the transmission queue was illegal and the NIC register
    //write request is denied.
    else if transmit_overlap = ILLEGAL_OVERLAP then
        return (i, false)
    //Checks overlap for the reception queue in a nearly identical way as for
    //the transmission queue.
    else
        word32: receive_overlap := type_of_cppi_ram_access_overlap(i, pa, i.oracle.rx0_active_queue)
        (i, bool: is_queue_secure) := is_queue_secure(i, val, RECEIVE)
        if receive_overlap = ZEROED_NDP_OVERLAP ∧ is_queue_secure then
            i := update_ρNIC_α_queue(i, val, RECEIVE, ADD)
            i.nic := write_nic_register(i.nic, pa, val)
            i := handle_potential_misqueue_condition(i, RECEIVE, pa, val)
            return (i, true)
        else        //receive_overlap = ILLEGAL_OVERLAP since α was true for pa.
            return (i, false)
```

B.4.2 Proof of Lemma V for cppi_ram_handler

This subsection describes how Lemma V can be proved for *cppi_ram_handler* when the NIC in the real model completes the processing of the transmission queue after the update of *tx0_active_queue* but before the transmission queue is extended.

How do the five different types of autonomous NIC transitions relate to each other?

- Initialization: Can only occur after both transmit and receive are idle (see nic_scheduler). Cannot run in parallel with teardown processes (see See checks on init_complete, init_step, transmit_teardown_step and receive_teardown_step in write_cpdma_soft_reset, write_tx_teardown and write_rx_teardown). Hence it runs alone in the NIC, if not a dead state is to be entered, which the current NIC configuration is not in since R and S holds and are transferred to the real initial state and those properties are not broken by this function.
- Transmit: Can run in parallel with the receive and receive teardown processes. Can only operate on buffer descriptors in tx0_active_queue (by R since tx0_active_queue includes all transmit buffer descriptors and they are not overlapped with the buffer descriptors in rx0_active_queue which includes all receive buffer descriptors).
- Transmit teardown: Can only occur after transmit is idle (see nic_scheduler). Can only operate on buffer descriptors in tx0_active_queue (by R since tx0_active_queue includes all transmit buffer descriptors and they are not overlapped with the buffer descriptors in rx0_active_queue which includes all receive buffer descriptors).
- Receive: Can run in parallel with the transmit and transmit teardown processes. Can only operate on buffer descriptors in rx0_active_queue (by R since rx0_active_queue includes all receive buffer descriptors and they are not overlapped with the buffer descriptors in tx0_active_queue which includes all transmit buffer descriptors).
- Receive teardown: Can only occur after receive is idle (see nic_scheduler). Can only operate on buffer descriptors in rx0_active_queue (by R since rx0_active_queue includes all receive buffer descriptors and they are not overlapped with the buffer descriptors in tx0_active_queue which includes all transmit buffer descriptors).

Hence transmit buffer descriptors are only affected by transmit and transmit teardown processes, and receive buffer descriptors are only affected by receive and receive teardown processes. Furthermore, the transmission and reception processes cannot run simultaneously as the corresponding teardown processes.

The initialization process is not active so forget about it. Since teardown processes are inactive, forget about them (lemma is needed that the following holds initialized ⇒ init_step = 0, tx0_tearingdown ⇒ transmit_step = 0 and rx0_tearingdown ⇒ process_received_frame_step = 0).

Now, the interesting question is: are transmit and receive autonomous NIC transitions independent? Since they access different buffer descriptors, yes, but still not completely independent. The reason is the following scenario: $o_0 \rightarrow_{\text{NIC\_MEMORY\_READ(0, 0)}} o_1 \rightarrow_{\text{NIC\_MEMORY\_WRITE(0, 0xFF)}} o_2$. What happens if these two transitions are reordered starting from the state o_0? $o_0 \rightarrow_{\text{NIC\_MEMORY\_WRITE(0, 0xFF)}} r_1 \rightarrow_{\text{NIC\_MEMORY\_READ(0, 0)}} o_2$ does not hold since the reception process overwrote the byte read by the transmission process. Therefore the reordered trace looks as follows: $o_0 \rightarrow_{\text{NIC\_MEMORY\_WRITE(0, 0xFF)}} r_1 \rightarrow_{\text{NIC\_MEMORY\_READ(0, 0xFF)}} o_2$. The end state o_2 is unaffected by the reordering since memory contains the same information.

However, the surrounding outside world outside the local computer system will get 0xFF instead of 0x00. Do we care about that? No! The reason is that we want to create a new real trace whose initial and final states are identical to the original trace, and such that the new trace can be matched by the ideal model. Then the initial and final states of the original trace are related to the initial and final states of the ideal trace and hence the original real trace (transition in $\rightsquigarrow$) is secure and is allowed behavior, which is what is wanted. What happens to the outside world is not interesting. Hence transmit and receive transitions can be reordered as wanted.

A key property when reordering transmit and receive transitions is that the NIC does not depend on memory content. If that was not the case, the end state after the reordering (o_2 in the example above) would not be the same which could make it impossible or very difficult to show that the final state of the complete reordered trace is identical to the final state of the original real trace. Luckily the NIC does not depend on memory content.

By a lemma and security invariant: initialized ⇒ init_step = 0, ¬tx0_tearingdown ⇒ transmit_step = 0, and ¬rx0_tearingdown ⇒ process_received_frame_step = 0. Hence only transmit and receive processes make transitions during the execution of write_cppi_ram that affect the results computed by write_cppi_ram.

Reordering original trace:

1. Transitions during execution of update_active_queue(TRANSMIT): Only autonomous transmit (process) NIC transitions as part of the transmission process of the NIC affect the result of the computation of this function call: The ownership bit of the buffer descriptors in tx0_active_queue. This bit only depends on autonomous transmit NIC transitions. This is the case since tx0_tearingdown is false, which means that the transmit teardown process is idle (meaning that it is not active and it is not waiting for the transmit process to end) and hence it cannot set the ownership and teardown bits (lemma: just before the function is called: tx0_tearingdown ⇒ transmit_step = 0 must hold), and because the queues pointed to by tx0/rx0_active_queue do not overlap themselves or each other (hence the ownership bit cannot be modified as a result of reception or reception teardown process, and the initialization process is idle but does not modify CPPI_RAM, and therefore these bits cannot be modified in an illegal way) which is a lemma (lemma: just before the function is called: all buffer descriptors in tx0/rx0_active_queue are isolated from each other and contain all used buffer descriptors of the NIC). Another lemma is that the function cannot break these lemmas since they must hold for the next

function call: the lemma is preserved since the next descriptor pointers are not modified and tx0_active_queue is only advanced passed released buffer descriptors.

Therefore only autonomous NIC transmit transitions affect the call to update_active_queue(TRANSMIT). Therefore, of the autonomous NIC transitions that occur during the execution of update_active_queue, the autonomous NIC transitions that affect the termination condition of update_active_queue are moved to occur before update_active_queue starts execute, and all other NIC transitions are moved to occur after update_active_queue terminates.

2. Transitions during execution of update_active_queue(RECEIVE): Only autonomous receive NIC transitions affect the computation of this function call. That is the case since the buffer descriptors operated on by the transmit and receive processes are disjunct: all active buffer descriptors are in the queues pointed to by tx0/rx0_active_queue and they do not overlap or each other. Again, what the outside world sees is Linux guest memory is not relevant.

   Therefore all autonomous receive NIC transitions that affect the termination condition of this call are moved to occur before the call to update_active_queue(TRANSMIT) (note TRANSMIT and not RECEIVE) and the rest of the autonomous receive NIC transitions are moved to occur after update_active_queue(RECEIVE). The autonomous transmit NIC transitions in step 1 that are moved after update_active_queue(TRANSMIT) are moved further back to after update_active_queue(RECEIVE). The internal order of the transmit transitions are always kept and likewise for receive transitions. However, the relative order of some transmit and receive transitions might get changed which affects what the outside world sees, but that is not relevant to the security of the system and is allowed by the ideal model.

3. If the write does not overlap an active buffer descriptor, then the write is performed on a word in CPPI_RAM that is not used by the NIC and hence the write and the NIC are independent. If there was an overlap, then the NIC is not accessed. Therefore all transitions as ordered according to step 2 that follow update_active_queue(RECEIVE) are independent of this code block and are moved to occur after it.

4. By cppi_ram_overlap lemma the autonomous NIC transitions moved to occur after the overlap check can be moved to occur after the call to type_of_cppi_ram_access_overlap.

5. There are three cases depending on the result from type_of_cppi_ram_access_overlap.

   A) Zeroed next descriptor pointer: By is_queue_secure lemma the autonomous NIC transitions moved to occur after the transmit queue overlap check can be moved to occur after this call. However, there are two cases depending on whether true or false is returned.

i. True: Since is_queue_secure returned true with value as argument it means that that queue does not overlap any active queue. update_ρ_{NIC}_$\varphi_{ACTIVE_CPPI_RAM}$_queue only accesses next descriptor pointer fields, in the transmit case, of the queue pointed to by value. Hence this call is independent of autonomous NIC transitions and therefore all autonomous NIC transitions that occur after is_queue_secure can be moved to occur after this call.

   Now the next descriptor pointer field of the last buffer descriptor of the transmit queue is written with the physical address of the new queue. If the autonomous NIC transitions occurring before this write have not emptied the transmit queue to be extended with the new one, then all autonomous NIC transitions occurring after is_queue_secure can be moved to occur after this write, without affecting those transitions compared to the original trace. Then write_cppi_ram terminates and all autonomous NIC transitions occurring during this scenario can be moved to either before or after this function and it can execute atomically.

   If the autonomous transmit NIC transitions occurring before this write has emptied the transmit queue to be extended, then they cannot be moved to occur after this write. If the transmit queue to be extended became empty between update_active_queue and this CPPI_RAM write in the original trace, then those autonomous transmit NIC transitions cannot be moved to occur either before update_active_queue or after this write. In the former case, tx0_active_queue and $\varphi_{ACTIVE\_CPPI\_RAM}$ (manipulated by update_active_queue) would have a different values compared to the original trace when comparing the final states of the handler traces. In the latter case the NIC would process the new queue and therefore end up in a different state compared to the original trace.

   Therefore the autonomous NIC transmit transitions in the original trace that occur after the autonomous NIC transmit transitions that affect the termination condition of update_active_queue(TRANSMIT) and before the write to CPPI_RAM are stuck between these to events. Immediately after the CPPI_RAM write, write_cppi_ram terminates.

B.5 Granularity of Specification Transitions

This section discusses which granularity the oracle transitions can have and why atomic transitions are desirable:

- Atomic oracle: At one extreme, an oracle transition can be defined to perform all operations of its exception handler.
- Coarse-grained oracle: Another alternative could be to define a transition to perform all oracle operations that occur between each NIC register access.

- Fine-grained oracle: At the other extreme, an oracle transition can be defined to perform one operation, such as assigning a value to a variable or computing an if-condition.

The following aspects are worth to consider when choosing the granularity of the oracle transitions:

- The number of definitions of the transitions rules and their complexity.
- Preservation of *S*.
- Difficulties in reordering the CPU and NIC transitions when creating a new sub-trace in the real model that shall be used to be matched by oracle transitions.

The following three subsections describe how well these three aspects are considered by the three transition granularities.

B.5.1 Complexity of Transitions

The more fine-grained the transitions are, the more complex they get:

- More transitions must be defined and what parts of an exception handler a transition include.
- The state needs to be extended with additional control information and the transitions with additional operations and side conditions. The reason is that the transitions that constitute a handler must be applied in correct order, which these additional operations and side conditions do by operating on the additional control information.
- If a complete if-then-else statement is not performed by a single transition, then parts of the then statement or else statement is performed by a different transition. This complicates the task described in the previous bullet.
- If a higher-level operation is split into several transitions, then it gets harder to analyze the outcome of that operation since a sequence of transitions must be considered, and not just a single one as is the case with an atomic oracle.

Also, if the oracle transitions are too fine-grained, the ideal model looses its value of specifying a valid design, since low-level details are included in the reasonings which takes focus from the actual operation. This would mean that it might not be too far away from proving everything on the real model alone.

To summarize, the lower the abstraction level is of transition rules, the more details and overhead information must be considered when defining the transitions. This aspect means an atomic oracle is best.

B.5.2 Preservation of SEC

There are two scenarios to consider when proving that the oracle transitions preserve *SEC*: the transition does not access a sensitive resource (*SEC* does not depend on it, like τ for instance), or the transition does access a sensitive resource

(*SEC* does depend on it). If a transition does not write resources mentioned by *SEC* (reads do not affect the value of *SEC*), the transition granularity does not really matter.

If a transition does write resources that *SEC* depends on, then certain properties of the value written must be known in order to prove that *SEC* is preserved. If the value written is computed by another transition, the proof gets a bit harder for lower-level transition rules because it must be considered which earlier transitions could have been applied, in which order they were applied, and what operations they performed. Hence, in this respect, atomic transitions are best.

B.5.3 Rescheduling of CPU and NIC

It is easier to create a new real model sub-trace that can be matched by a sub-trace of the ideal model if the ideal sub-trace is allowed to have NIC transitions interleaved with non-atomic specification transitions. However, since it has been reasoned for most execution scenarios for most NIC register write request handlers that Lemma V can be proved for them, the conclusion is that atomic specification transitions can be managed.

With the reasoning in the previous subsections, atomic specification transitions seem to be the best solution, and has therefore been used in the work presented in this thesis. However, it is unknown at this time how difficult it is to prove in HOL4 that reordered real model sub-traces can be matched by atomic specification transitions. Hence, the use of non-atomic specification transitions should not be ruled out after all.

Appendix C Model of Network Interface Controller

This appendix describes the NIC model by means of the pseudocode notation described in Appendix A, including comments. To save space and due to the similarities of the transmission and reception related processes, are the reception and reception teardown processes omitted. The differences are that the reception process issues memory write requests instead of memory read requests, and that the reception and reception teardown processes sets additional fields in the buffer descriptors after a frame has been processed. It is upon this formal model that it has been reasoned that the NIC register write request handlers preserve the formal definition of *S* in Appendix D.

First is a description given of the modeled NIC registers, and second is a list presented of assumptions made about the content of certain NIC registers, some of which are not modeled. Following that are the NIC state and the pseudocode that specifies the four types of NIC transitions defined: NIC register reads, NIC register writes, autonomous NIC transitions and memory read request replies, where the latter two are specified by the NIC scheduler and the NIC processes. The final section describes the registers that are necessary to model if the interrupts that Linux uses are to be modeled accurately. The current NIC model asserts interrupts correctly but does so non-deterministically to simulate both behaviors of enabled and disabled interrupts.

C.1 NIC Registers in NIC Model

The registers are modeled as variables storing bit strings and functions mapping bit strings to bit strings. All physical NIC registers are allocated 32 bits but some NIC registers do not use all of their 32 bits. Such registers are modeled as bit strings with fewer bits. The modeled registers are the following ones:

- DMACONTROL: Consists of 16 bits but only bit one (RX_OWNERSHIP) and two (RX_OFFLEN_BLOCK) are relevant for the purpose of the NIC model. These two bits affect how the NIC manipulates receive buffer descriptors in CPPI_RAM. The operations that this register causes are not modeled. The NIC enters a dead state if any of the bits of this register are set by the CPU.
- CPDMA_SOFT_RESET: Consists of one bit which is set to one by the CPU to reset the NIC DMA hardware, and is cleared by the NIC when the reset is complete. This causes the initialization process to be activated when the transmission and reception processes have finished the handling of their current frames (if any).
- RX_BUFFER_OFFSET: Consists of 16 bits. This register determines how many leading bytes that shall be skipped when storing the first part of a received frame in a data buffer in memory. For instance, if it is three then the first three bytes of the data buffer are not used to store the received

frame. In effect, received frames get their lengths extended by the amount stored in this register, which affects memory accesses.

- CPPI_RAM: Models the memory area of the NIC that holds the buffer descriptors. This memory area consists of 2^{13} bytes and is modeled as a function that maps bit strings of length 13 to bytes. The argument is the offset address relative to the start of the first physical address of CPPI_RAM (0x4A102000). Writing this register can potentially affect the already active transmission and reception processes.
- TX0_HDP. Set to the physical address of the head of a buffer descriptor queue in CPPI_RAM to activate the transmission process. The written value is the physical address of the buffer descriptor as seen from the perspective of the complete address space of the computer, and not as an offset to the start of the CPPI_RAM register. When transmission is finished is this register zeroed.

  This register is modeled in a peculiar way because the modifications of this register during transmissions are not completely specified. When TX0_HDP is written by the CPU, the new value is stored in a NIC model variable called *transmit_current_bd* and TX0_HDP is given an arbitrary non-deterministically chosen non-zero value. The chosen value must be non-zero because a zero value means that the transmission process is idle. However, TX0_HDP is set to zero after a reset operation, if the CPU writes that value because that behavior is specified.
- RX0_HDP: Set to the head of a buffer descriptor queue in CPPI_RAM that is used to store received frames. Writing RX0_HDP does not activate the reception process, but it allows the NIC scheduler to non-deterministically decide that a new frame has arrived which in turn activates the reception process. RX0_HDP is zeroed when all buffer descriptors in the reception queue have been consumed by the reception process.

  As in the case with TX0_HDP, this register is also modeled as being written with a non-deterministically chosen non-zero value when the CPU wants to give the NIC a new reception queue and enable reception. The variable *receive_current_bd* is set to the value that the CPU wanted to write. However, by reading the RX0_HDP register on the actual hardware during receptions, RX0_HDP seems to be advanced to the next buffer descriptor in the queue when the NIC has completed the processing of its current buffer descriptor.
- TX0_CP: The content of this register can only be set by the NIC. Writes by the CPU are not stored in this register but are used to acknowledge frame transmission completion and transmission teardown interrupts. The transmission process sets this register to the address of the last buffer descriptor of the most recently transmitted frame when that frame transmission is complete, or to 0xFFFFFFFC when a teardown process has completed. Such writes make the NIC assert an interrupt and which. These interrupts are deasserted by the NIC when the CPU writes the value

currently stored in this register. Other written values by the CPU are ignored by the NIC.

- RX0_CP: Used in an identical way as TX0_CP but with respect to reception instead of transmission.
- TX_TEARDOWN: Consists of three bits and is always read as zero. Software writes the ID of the transmission DMA channel that is to be teared down. Since only channel zero is modeled, does the NIC enter a dead state if any other value is written by the CPU. Writing this register with the value zero allows the NIC scheduler to schedule the transmission teardown process, as soon as the transmission process has finished the processing of its current frame. The transmissions of following frames are canceled.
- RX_TEARDOWN: Is similar to the TX_TEARDOWN register but disables reception, and hence the possibility to store future incoming frames in memory.

The NIC state variables that correspond to hardware registers are written in capital letters, using the same names as given by the NIC hardware specification, while all other variables are written with small letters. Register writes by the CPU are modeled by the function *write_nic_register* which takes as arguments the NIC state, the physical address of the register to write and the value to write. Register reads are modeled by the function *read_nic_register* which takes as its arguments the NIC state and the physical address of the register to read, and returns the input NIC state, since NIC register reads have no side effects, and the read register value.

C.2 Assumptions

The operation specified by the NIC model depends on some assumptions about the content of certain NIC registers, several of which have not been modeled. Some assumptions are made to make the NIC model useful without including the registers in the model, and some assumptions are made to include more behavior of the NIC to increase the set of potentially accessed memory addresses. No assumption is in conflict with the operation of Linux. The assumptions are:

- TX_CONTROL and RX_CONTROL: The TX_EN and RX_EN bits are set to enable the DMA controller for transmission and reception, respectively. If they are cleared, then the NIC does not access memory.
- MACCONTROL:
  - The RX_CMF_EN bit is set to enable transfers of received MAC control frames to memory. This assumption allows the NIC to write certain bits of buffer descriptors that would otherwise not be made.
  - The RX_CSF_EN bit is set to allow reception of frames that are shorter than 64 bytes. This assumption allows the NIC to write data buffers that are smaller than 64 bytes.
  - The RX_CEF_EN bit is set to allow reception of frames that contain errors and that are longer than specified in the non-modeled register RX_MAXLEN. This assumption allows frames of arbitrary size to be

stored in memory, and it also allows the NIC to write certain buffer descriptor fields.

- DMACONTROL:
    - The RX_CEF bit is set to allow buffer overruns. This property enables the NIC to store received frames that do not fit in the data buffers, as specified by the buffer descriptors in the receive queue, to the extent the frames fit. This assumption allows the NIC to write the overrun bit in buffer descriptors. However, the modeling of DMACONTROL assumes this bit is zero. The meaning of this is that the NIC model acts as if this bit is set, but the content of DMACONTROL acts as if this bit is cleared. That is, the behavior is modeled, but not the content of this register with respect to this bit. Linux does not set this bit. This assumption is made to allow additional CPPI_RAM modifications to increase the set of potential memory accesses.
    - The RX_OFFLEN_BLOCK bit is cleared to make the NIC write the buffer offset and buffer length fields of the third word of receive buffer descriptors. This reflects the modeled behavior of the NIC. That is, the NIC model acts as if this bit is cleared.
    - The RX_OWNERSHIP bit is cleared to make the NIC clear the ownership bit in receive buffer descriptors when the NIC is done with a set of buffer descriptors related to a received frame. The NIC model describes the operation of the NIC as if this bit is cleared.

C.3 Definition of State of NIC

The variables of the NIC model shall be initialized as described by the comments in the definition of the NIC state. The hardware registers are initialized to their reset values, but they are unspecified for CPPI_RAM, which shall therefore be initialized non-deterministically. An initialized NIC state corresponds to a state where the NIC has just been powered on. The definition follows.

```
nic_state = (                                   //The type of the state of the NIC.
    bool: dead_state,                           //Initialized to false. True if the current NIC state is undefined.
    bool: interrupt,                            //Initialized to false. True if a frame completion interrupt is asserted.
    nic_regs: regs,                             //The component containing all modeled NIC registers.
    init_state: init_p,                         //The component of the initialization process.
    transmit_state: tx_p,                       //The component of the transmission process.
    receive_state: rx_p,                        //The component of the reception process.
    transmit_teardown_state: tx_td_p,           //The component of the transmission teardown process.
    receive_teardown_state: rx_td_p             //The component of the reception teardown process.
)
nic_regs = (                                    //The type of the record that contains all modeled NIC registers.
    word16: DMACONTROL,                         //Reset value is zero.
    word1: CPDMA_SOFT_RESET,                    //Reset value is zero.
    word16: RX_BUFFER_OFFSET,                   //Reset value is zero.
    word13 → word8: CPPI_RAM,                   //Reset value is unspecified. Can be chosen non-deterministically.
    word32: TX0_HDP,                            //Reset value is zero. Its value during transmission is undefined.
    word32: RX0_HDP,                            //Reset value is zero. Its value during reception is undefined.
    word3: TX_TEARDOWN,                         //Reset value is zero.
    word3: RX_TEARDOWN,                         //Reset value is zero.
    word32: TX0_CP,                             //Reset value is zero.
    word32: RX0_CP                              //Reset value is zero.
)
init_state = (                                  //The type of the state of the initialization process.
    bool: init_complete,                        //Initialized to false. True if NIC has been initialized and init_step = 0.
```

```
        bool: tx0_hdp_initialized,                 //Initialized to false. True if TX0_HDP has been zeroed after a reset.
        bool: rx0_hdp_initialized,                 //Initialized to false. True if RX0_HDP has been zeroed after a reset.
        bool: tx0_cp_initialized,                  //Initialized to false. True if TX0_CP has been zeroed after a reset.
        bool: rx0_cp_initialized,                  //Initialized to false. True if RX0_CP has been zeroed after a reset.
        nat: init_step                             //Initialized to zero. 0 if idle, 1 to do reset, 2 if initialize registers.
)
transmit_state = (                                 //The type of the state of the transmission process.
        nat transmit_step,                         //Initialized to zero. Specifies next step function to apply.
        nat transmit_read_buffer_descriptor_bytes, //Need not be initialized. Number of read bytes of current descriptor.
        word32 transmit_current_bd,                //Need not be initialized. Physical address of current buffer descriptor.
        word32 transmit_current_bd_value_word0,    //Need not be initialized. Word 0 of the current buffer descriptor.
        word32 transmit_current_bd_value_word1,    //Need not be initialized. Word 1 of the current buffer descriptor.
        word32 transmit_current_bd_value_word2,    //Need not be initialized. Word 2 of the current buffer descriptor.
        word32 transmit_current_bd_value_word3,    //Need not be initialized. Word 3 of the current buffer descriptor.
        word32 transmit_sop_bd_physical_address,   //Need not be initialized. Physical address of SOP descriptor of frame.
        word32 transmit_eop_bd_physical_address,   //Need not be initialized. Physical address of EOP descriptor of frame.
        word32 transmit_next_buffer_byte_address,  //Need not be initialized. Physical address of next byte to transmit.
        bool transmit_expect_sop,                  //Need not be initialized. True if next descriptor shall be SOP.
        word32 transmit_buffer_length,             //Need not be initialized. Number of bytes left to transmit of frame.
        word11 transmit_sop_packet_length,         //Need not be initialized. The packet length field of the SOP descriptor.
        nat transmit_sum_buffer_length,            //Need not be initialized. The sum of all buffer length fields of frame.
        bool memory_request,                       //Initialized to false. True if NIC waits for memory read reply.
        bool interrupt                             //Initialized to false. True if the NIC asserts a tx or td interrupt.
)
transmit_teardown_state = (                        //The type of the state of the transmission teardown process.
        nat transmit_teardown_step                 //Initialized to zero. Specifies next step function to apply. 0 when idle.
)
```

C.4 Reads of NIC Registers

```
/*
 *      @nic: The NIC state which defines the content of the NIC register to read.
 *
 *      @physical_address: word32. The physical address of the NIC register to read.
 *
 *      If @physical address is not word aligned, then does the NIC enter a dead
 *      state. Also, if the register at @physical_address is not modeled, then is a
 *      non-deterministically 32-bit value returned.
 */
(nic_state, word32): read_nic_register(nic_state: nic, word32: physical_address)
        if physical_address[1:0] ≠ 0 then //Checks word alignment.
                nic.dead_state := true
                return (nic, 0)
        else if physical_address = 0x4A10_0808 then
                return (nic, 0)    //TX_TEARDOWN is read as zero.
        else if physical_address = 0x4A10_0818 then
                return (nic, 0)    //RX_TEARDOWN is read as zero.
        else if physical_address = 0x4A10_081C then
                return (nic, nic.regs.CPDMA_SOFT_RESET)
        else if physical_address = 0x4A10_0820 then
                return (nic, nic.regs.DMACONTROL)
        else if physical_address = 0x4A10_0828 then
                return (nic, nic.regs.RX_BUFFER_OFFSET)
        else if physical_address = 0x4A10_0A00 then
                return (nic, nic.regs.TX0_HDP)
        else if physical_address = 0x4A10_0A20 then
                return (nic, nic.regs.RX0_HDP)
        else if physical_address = 0x4A10_0A40 then
                return (nic, nic.regs.TX0_CP)
        else if physical_address = 0x4A10_0A60 then
                return (nic, nic.regs.RX0_CP)
        //Makes sure that the whole word is within CPPI_RAM by checking that the third
        //byte is within the upper limit. This is done by subtracting the upper limit by three.
        else if 0x4A10_2000 ≤ physical_address ∧ physical_address < 0x4A10_4000 - 0x3 then
                return (nic, nic.regs.CPPI_RAM(physical_address - 0x4A10_2000 + 3) ::
                                nic.regs.CPPI_RAM(physical_address - 0x4A10_2000 + 2) ::
                                nic.regs.CPPI_RAM(physical_address - 0x4A10_2000 + 1) ::
                                nic.regs.CPPI_RAM(physical_address – 0x4A10_2000))
        //Non-modeled NIC registers returns an arbitrary value.
        else
                return (nic, choice_non-deterministically(word32))
```

C.5 Writes to NIC Registers

```
/*
 *     @nic: The NIC state to operate on.
 *
 *     @physical_address: Physical address of the NIC register to write.
 *
 *     @value: Value to write to the NIC register at @physical_address.
 *
 *     If the given NIC state is not in a dead state, @physical_address is 32-bit
 *     word aligned, and no non-modeled HDP register is written with a non-zero
 *     value, then is @value written into that register. Otherwise does the NIC
 *     either nothing or enter a dead state. The updated NIC state as a result of
 *     the NIC register write is returned.
 */
nic_state: write_nic_register(nic_state: nic, word32: physical_address, word32: value)
    if ¬nic.dead_state then
        if physical_address[1:0] ≠ 0 then  //Checks word alignment.
            nic.dead_state := true
        else if ((0x4A10_0A00 < physical_address ∧ physical_address < 0x4A10_0A20) ∨
                (0x4A10_0A20 < physical_address ∧ physical_address < 0x4A10_0A40)) ∧
                value ≠ 0 then    //Writing a non-modeled HDP register with a non-zero value.
            nic.dead_state := true
        else
            if physical_address = 0x4A10_0808 then
                nic := write_tx_teardown(nic, value[2:0])
            else if physical_address = 0x4A10_0818 then
                nic := write_rx_teardown(nic, value[2:0])
            else if physical_address = 0x4A10_081C then
                nic := write_cpdma_soft_reset(nic, value[0])
            else if physical_address = 0x4A10_0820 then
                nic := write_dmacontrol(nic, value[15:0])
            else if physical_address = 0x4A10_0828 then
                nic := write_rx_buffer_offset(nic, value[15:0])
            else if physical_address = 0x4A10_0A00 then
                nic := write_tx0_hdp(nic, value)
            else if physical_address = 0x4A10_0A20 then
                nic := write_rx0_hdp(nic, value)
            else if physical_address = 0x4A10_0A40 then
                nic := write_tx0_cp(nic, value)
            else if physical_address = 0x4A10_0A60 then
                nic := write_rx0_cp(nic, value)
            else if 0x4A10_2000 ≤ physical_address ∧ physical_address < 0x4A10_4000 then
                nic := write_cppi_ram(nic, physical_address, value)
            //Accesses to non-modeled NIC registers have no effect.

    return nic

/*
 *     If the initialization or transmission teardown processes are active or the
 *     value to write is not zero (the transmission DMA channel to tear down),
 *     then writing this register is undefined (dead state entered), since only
 *     channel zero is modeled or otherwise unspecified behavior in the NIC
 *     specification. Otherwise the value to write takes effect by setting
 *     nic.tx_td_p.transmit_teardown_step to one. The new NIC state is returned.
 */
nic_state: write_tx_teardown(nic_state: nic, word3: value)
    if ¬nic.init_p.init_complete ∨ nic.tx_td_p.transmit_teardown_step > 0 then
        nic.dead_state := true
    else
        if value ≠ 0 then
            nic.dead_state := true
        else
            //The transmit teardown process is to be started as soon as the
            //current frame under transmission has been sent.
            nic.tx_td_p.transmit_teardown_step := 1

    return nic

/*
 *     Writing this register if any of the following conditions hold results in an
 *      undefined operation (dead state entered):
 *      *Initialization has never been performed and the bit value to write is not
 *       one. Actually this is not undefined by true hardware but forces the
 *       software to write a one the first time this register is written. Does not
 *       make a difference since writing a zero to this register has no effect.
 *      *Initialization has already been started (init_step > 0).
 *      *Initialization has been performed and the initialization process is
 *       inactive, the bit value to write is one, and the transmit and/or receive
 *       teardown processes are in progress.
 *
 *      If the write is defined, then init_step is set to one to make the
 *      initialization process of the NIC active, and if needed it waits until the
 *      transmission and reception processes have finished the processing of their
 *      currently processed frames. The boolean variables that keep track of the
 *      initialization of the HDP and CP registers are also falsified.
 *
 *      Initialization in this context involves both CPDMA logic reset and zeroing
 *      of the HDP and CP registers.
 */
nic_state: write_cpdma_soft_reset(nic_state: nic, word1: value)
        //Initialization is not done or is under progress.
        if ¬nic.init_p.init_complete then
                //Initialization has never been done: ¬init_complete ∧ init_step = 0.
                if value = 1 ∧ nic.init_p.init_step = 0 then
                        //Writes CPDMA_SOFT_RESET to indicate that the reset is under operation.
                        nic.regs.CPDMA_SOFT_RESET := 1
                        //First step of the initialization process.
                        nic.init_p.init_step := 1
                        //Falsifies all initialization flags that tracks the initialization
                        //of the HDP and CP registers.
                        nic.init_p.init_complete := false
                        nic.init_p.tx0_hdp_initialized := false
                        nic.init_p.rx0_hdp_initialized := false
                        nic.init_p.tx0_cp_initialized := false
                        nic.init_p.rx0_cp_initialized := false
                else                                            //Reset is already under progress.
                        nic.dead_state := true
        else                            //Initialization has been done and is inactive.
                //Activating reset while teardown processes are active is unspecified.
                if value = 1 ∧ (transmit_teardown_step > 0 ∨ receive_teardown_step > 0) then
                        nic.dead_state := true
                else if value = 1 then                  //Activating the reset process.
                        nic.regs.CPDMA_SOFT_RESET := 1
                        nic.init_p.init_step := 1
                        nic.init_p.init_complete := false
                        nic.init_p.tx0_hdp_initialized := false
                        nic.init_p.rx0_hdp_initialized := false
                        nic.init_p.tx0_cp_initialized := false
                        nic.init_p.rx0_cp_initialized := false
                //If value is zero, nothing needs to be done since init_complete was
                //true when this function was applied which means that CPDMA_SOFT_RESET
                //is already zero. Writing zero has no effect.

        return nic

/*
 *      Writes the TX0_HDP register. This activates transmission DMA channel zero.
 *
 *      This is an undefined operation if any of the following conditions hold:
 *      *The initialization process is active, the reset operation is complete, but
 *       the value to write is not zero.
 *      *The initialization process is active and the reset operation is not complete.
 *      *Initialization is complete and the TX0_HDP register is not equal to zero.
 *      *Initialization is complete and the transmit teardown process is active.
 */
nic_state: write_tx0_hdp(nic_state: nic, word32: bd_physical_address)
        //Initialization is not complete.
        if nic.init_p.init_complete = false then
                //Initialization is in progress, the reset operation is performed and
                //the value to write is zero: initialization conditions are satisfied.
                if bd_physical_address = 0 ∧ nic.init_p.init_step = 2 then
                        nic.regs.TX0_HDP := 0                   //bd_physical_address is zero.
```

```
            nic.tx_p.transmit_current_bd := 0       //Synchronized with TX0_HDP.
            nic.init_p.tx0_hdp_initialized := true  //TX0_HDP is now initialized.

            //Initialization complete.
            if nic.init_p.tx0_hdp_initialized ∧ nic.init_p.rx0_hdp_initialized ∧
                    nic.init_p.tx0_cp_initialized ∧ nic.init_p.rx0_cp_initialized then
                nic.init_p.init_complete := true
                nic.init_p.init_step := 0
        else
            //If the initialization process has never been performed and is
            //inactive, the reset operation has not finished or the value to
            //write is not zero, then the dead state is entered.
            nic.dead_state := true
    else
        //Initialization has been performed.
        //Writing TX0_HDP when it is not zero is an error according to the NIC
        //specification, and when it is zero, is the tranmission process idle
        //since it clears HDP as its last operation.
        if nic.regs.TX0_HDP ≠ 0 then
            nic.dead_state := true
        //If the transmission teardown process is active, then the dead state
        //is entered, even though TX0_HDP is zero.
        else if nic.tx_td_p.transmit_teardown_step > 0 then
            nic.dead_state := true
        else if bd_physical_address ≠ 0 then
            //TX0_HDP is set to an unknown value since the specification does
            //not state anything about how its value changes during transmission.
            //If TX0_HDP is zero, then it indicates that the transmission
            //process is idle, and therefore must the value to assign it be
            //non-zero. In such case it is incremented by one.
            nic.regs.TX0_HDP := choice_non-deterministically(word32)
            if nic.regs.TX0_HDP = 0 then nic.regs.TX0_HDP := 1
            //The transmit_current_bd variable is set to the buffer descriptor's
            //physical address.
            nic.tx_p.transmit_current_bd := bd_physical_address
            //The transmission process as been re-activated only if it is
            //terminated, including issuing interrupt. Otherwise step 7 of the
            //transmission process takes care of re-initiating it unless a
            //pending reset or teardown operation is pending.
            if nic.tx_p.transmit_step = 0 then
                nic.tx_p.transmit_step := 1
            //A SOP buffer descriptor is expected to be the first buffer
            //descriptor in the transmit queue.
            nic.tx_p.transmit_expect_sop := true
        //If the TX0_HDP register is already zero and is to be set to zero,
        //then nothing happens.

    return nic

/*
 *  Has the same initialization requirements as the HDP registers. If they are
 *  not satisfied then a dead state is entered. If initialization has been
 *  done, then nothing happens since the hardware only deasserts potential
 *  interrupts, which are not modeled. Since it is unknown how the
 *  initialization of the CP registers affect interrupts, they are deasserted
 *  non-deterministically.
 */
nic_state: write_tx0_cp(nic_state: nic, word32: bd_physical_address)
    if ¬nic.init_p.init_complete then
        if bd_physical_address = 0 ∧ nic.init_p.init_step = 2 then
            nic.regs.TX0_CP := 0                    //bd_physical_address is zero.
            nic.init_p.tx0_cp_initialized := true

            if choice_non-deterministically(bool) then
                nic.tx_p.interrupt := false
                if ¬nic.rx_p.interrupt then
                    nic.interrupt := false

            if nic.init_p.tx0_hdp_initialized ∧ nic.init_p.rx0_hdp_initialized ∧
                    nic.init_p.tx0_cp_initialized ∧ nic.init_p.rx0_cp_initialized then
                nic.init_p.init_complete := true
                nic.init_p.init_step := 0
        else
            nic.dead_state := true

    //Otherwise init_complete = true and initialization is complete and the
    //completion pointer register can be written to any value. The hardware
    //does ignore the write though, except for interrupts which are deasserted
    //when the right CP value is written to the same value of its content.
    if nic.regs.TX0_CP = bd_physical_address then
        nic.tx_p.interrupt := false
        if ¬nic.rx_p.interrupt then
            nic.interrupt := false

    return nic

/*
 *   Writes a 32-bit word into CPPI_RAM in little endian order: Less significant
 *   bytes of the 32-bit word is written at a lower address. This must be
 *   consistent with how bytes are read from CPPI_RAM, for instance when reading
 *   a buffer descriptor. Intuitively this seems to be consistent with the NIC
 *   specification, but it does not state this little endian order explicitly.
 */
nic_state: write_cppi_ram(nic_state: nic, word32: cppi_ram_physical_address, word32: bd_word)
    nic.regs.CPPI_RAM(cppi_ram_physical_address - 0x4A10_2000) := bd_word[7:0]
    nic.regs.CPPI_RAM(cppi_ram_physical_address - 0x4A10_2000 + 1) := bd_word[15:8]
    nic.regs.CPPI_RAM(cppi_ram_physical_address - 0x4A10_2000 + 2) := bd_word[23:16]
    nic.regs.CPPI_RAM(cppi_ram_physical_address - 0x4A10_2000 + 3) := bd_word[31:24]

    return nic
```

C.6 Automata Scheduler

```
//mem_req is the type of memory read and write requests, and for reads also their replies.
mem_req = (
    bool: valid,                //True if and only if this object actually represents a memory request.
    bool: read,             //True if and only if the request corresponds to a memory read.
    word32: address,            //32-bit physical address of the location to read or write in memory.
    word8: value                //The byte to store or that contains the read memory byte.
)

//no_mem_req is a constant that is used by the functions that do not return memory requests.
mem_req: no_mem_req := (false, false, 0, 0)

/*
 *   This function is non-deterministically selecting the next NIC process that
 *   shall perform a transition. It represents an autonomous transition of the
 *   NIC. If no such process can perform a transition, then the returned NIC
 *   state is identical to the argument and no memory request is returned.
 *
 *   This function determines the next transition by forming a set P consisting
 *   all processes that can make a transition in the given NIC state. The
 *   processes are represented by the following symbolic constants:
 *   -init: The initialization process.
 *   -transmit: The transmission process.
 *   -receive_new_frame: A new frame is received and the receive frame process
 *    can make its first transition.
 *   -receive: The reception process.
 *   -transmit_teardown: The transmission teardown in progress.
 *   -receive_teardown: The reception teardown in progress.
 *
 *   If any of these processes cannot make a transition in the current state,
 *   then that process is not in the set P.
 *
 *   To make a transition a process is selected non-deterministically from P. A
 *   function is then applied for the selected process that determines the exact
 *   transition that process shall perform (determined by the step variable of
 *   the selected process), except receive_new_frame which always does the same
 *   thing. All these transitions are autonomous. That is, they occur internally
 *   in the NIC independent of the CPU and system bus.
 *
 *   Takes a NIC state as argument and returns:
 *   -An updated state according to an autonomous transition.
 *     -A memory read or write request.
 *     -A flag indicating whether the NIC currently raises an interrupt.
 */
(nic_state, mem_req, bool): nic_scheduler(nic_state: nic)
    if nic.dead_state then
        return (nic, no_mem_req, nic.interrupt)

    {{init, transmit, receive_new_frame, receive, transmit_teardown, receive_teardown}}: P := {}

    //The reset operation occurs at frame boundaries: the reset bit is set
    //and the currently transmitted and received frames are completely
    //processed.
    if nic.init_p.init_step = 1 ∧ nic.tx_p.transmit_step = 0 ∧ nic.rx_p.receive_step = 0 then
        P := P ∪ {init}

    //If the transmission process waits for a memory read reply, then it
    //cannot make a transition.
    if nic.tx_p.transmit_step > 0 ∧ ¬nic.tx_p.memory_request then
        P := P ∪ {transmit}

    //A new frame can be received if all of the following conditions hold:
    //-No frame is processed by the reception process.
    //-There is a receive buffer descriptor queue.
    //-The receive teardown process is inactive.
    //-The initialization process has been completed and is not operating.
    if nic.rx_p.receive_step = 0 ∧ nic.rx_p.receive_current_bd ≠ 0 ∧
            nic.rx_td_p.receive_teardown_step = 0 ∧ nic.init_p.init_complete then
        P := P ∪ {receive_new_frame}

    if nic.rx_p.receive_step > 0 then
        P := P ∪ {receive}

    //Teardown can only be executed when its operation has been requested
    //and the related transmission or reception process is inactive.
    if nic.tx_td_p.transmit_teardown_step > 0 ∧ nic.tx_p.transmit_step = 0 then
        P := P ∪ {transmit_teardown}

    if nic.rx_td_p.receive_teardown_step > 0 ∧ nic.rx_p.receive_step = 0 then
        P := P ∪ {receive_teardown}

    //Selects a process non-deterministically.
    {init, transmit, receive_new_frame, receive, transmit_teardown, receive_teardown}: transition
    transition := choice_non-deterministically(P)
    mem_req: memory_request := no_mem_req

    if transition = init then
        (nic, memory_request) := init(nic)
    else if transition = transmit then
        (nic, memory_request) := transmit_frame(nic)
    else if transition = receive_new_frame then
        (nic, memory_request) := receive_step0(nic)
    else if transition = receive then
        (nic, memory_request) := receive_frame(nic)
    else if transition = transmit_teardown then
        (nic, memory_request) := transmit_teardown(nic)
    else if transition = receive_teardown then
        (nic, memory_request) := receive_teardown(nic)

    return (nic, memory_request, nic.interrupt)
```

C.7 Initialization Automaton

```
/*
 *     The following function perform the hardware reset step, which is the
 *     first part of the initialization process. The second part is the
 *     initialization of the HDP and CP registers, which is done by writing NIC
 *     registers after a performed reset operation, without any other intervening
 *     NIC register writes.
 *
 *     Since the NIC specification does not state what the reset operation does,
 *     except from clearing the least significant bit of the CPDMA_SOFT_RESET
 *      register when it is done, no other hardware work is done than just that.
 *      init_step is also set to two to put the NIC into a state that indicates
 *      that the HDP and CP registers can now be initialized by writing them to
 *      zero. When they are initialized, init_step is set to zero, and
 *      init_complete to true.
 */
(nic_state, mem_req): init(nic_state: nic)
        nic.regs.CPDMA_SOFT_RESET := 0
        nic.init_p.init_step := 2
        return (nic, no_mem_req)
```

C.8 Transmission Automaton

```
/*
 *      These functions specify the transitions of the transmission process. This
 *      process is activated when TX0_HDP is set to a buffer descriptor queue (at
 *      which time nic.tx_p.transmit_step is set to 1). The transmission process
 *      terminates when all buffer descriptors in the transmission queue have been
 *      processed and the corresponding frames have been sent (at which time
 *      nic.tx_p.transmit_step is set to 0).
 *
 *      The transmission process is divided into eight step functions. Each frame
 *      transmission consists of applying all these step functions. The first five
 *      step functions processes the current buffer descriptor which contains one
 *      part of the frame under transmission. When all buffer descriptors have been
 *      processed as part of one frame transmission, post-processing begins with
 *      step 6, and ends at step 8. If additional frames exist in the transmission
 *      queue, then step 1 is performed again. Otherwise the transmission process
 *      terminates.
 *
 *      A summary of how the transmission process is performed:
 *      1.      Check that the current buffer descriptor is correctly located in
 *              CPPI_RAM.
 *      2.      Read the bytes of the current buffer descriptor from CPPI_RAM.
 *      3.      Check that the buffer descriptor is correctly initialized.
 *      4.      Issue memory read requests for the bytes of the associated data buffer.
 *              The replies are given by the framework by applying the NIC model
 *              function memory_request_byte_reply with the reply as an argument.
 *      5.      Check if the current buffer descriptor is the last buffer descriptor of
 *              the current frame under transmission. If not apply step 1, otherwise
 *              step 6.
 *      6.      If the EOP buffer descriptor was the last one, the EOQ bit is set of
 *              the EOP buffer descriptor.
 *      7.      Clears the ownership bit of the SOP buffer descriptor of the
 *              transmitted frame. If the EOP buffer descriptor was the last one, then
 *              TX0_HDP is cleared.
 *      8.      Sets TX0_CP to the physical address of the current buffer descriptor,
 *              which is the last one of the transmitted frame.
 */

/*
 *      Applied by the NIC scheduler when the transmission process shall perform
 *      the next transition of the NIC. It identifies and applies the step function
 *      that performs the actual transition.
 */
(nic_state, mem_req): transmit_frame(nic_state: nic)
        //If a transmission transition is to be performed when the transmission
        //process is waiting for a memory request there is a model bug.
        if nic.tx_p.memory_request then
                nic.dead_state := true
                return (nic, no_mem_req)
        else if nic.tx_p.nic.tx_p.transmit_step = 1 then
                return transmit_step1(nic)
        else if nic.tx_p.nic.tx_p.transmit_step = 2 then
                return transmit_step2(nic)
        else if nic.tx_p.nic.tx_p.transmit_step = 3 then
                return transmit_step3(nic)
        else if nic.tx_p.nic.tx_p.transmit_step = 4 then
                return transmit_step4(nic)
        else if nic.tx_p.nic.tx_p.transmit_step = 5 then
                return transmit_step5(nic)
```

```
    else if nic.tx_p.nic.tx_p.transmit_step = 6 then
        return transmit_step6(nic)
    else if nic.tx_p.nic.tx_p.transmit_step = 7 then
        return transmit_step7(nic)
    else if nic.tx_p.nic.tx_p.transmit_step = 8 then
        return transmit_step8(nic)
    else
        //If a transmission transition is to be performed but the transmission
        //process is inactive or the current transition has an undefined
        //number, then this is a model bug.
        nic.dead_state := true
        return (nic, no_mem_req)

/*
 *  Checks that the current buffer descriptor is 32-bit word aligned and
 *  completely located in CPPI_RAM, and initializes some model variables, and
 *  then (since these two former operations are not considered to do any
 *  hardware work) applies step function two.
 */
(nic_state, mem_req): transmit_step1(nic_state: nic)
    if nic.tx_p.transmit_current_bd[1:0] ≠ 0 ∨ nic.tx_p.transmit_current_bd < 0x4A10_2000 ∨
            0x4A10_4000 - 0xF ≤ nic.tx_p.transmit_current_bd then
        nic.dead_state := true
        return (nic, no_mem_req)
    else
        //Zero bytes have been read of the current buffer descriptor.
        nic.tx_p.transmit_read_buffer_descriptor_bytes := 0
        //Initializes the variables that hold the buffer descriptor words.
        nic.tx_p.transmit_current_bd_value_word0 := 0
        nic.tx_p.transmit_current_bd_value_word1 := 0
        nic.tx_p.transmit_current_bd_value_word2 := 0
        nic.tx_p.transmit_current_bd_value_word3 := 0

        //Step function 2 shall be applied next.
        nic.tx_p.transmit_step := 2
        return transmit_step2(nic)

/*
 *  Each application of this function reads the next byte of the current buffer
 *  descriptor from CPPI_RAM. When the last byte has been read transmit_step is
 *  set to 3 to make step 3 perform the next transition the next time the
 *  transmission process is scheduled.
 *
 *  A byte is inserted into the transmit_current_bd_value_wordx variable by
 *  shifting it into its right position and then doing bitwise OR with it and
 *  the current value of transmit_current_bd_value_wordx variable and storing
 *  the result in that same variable.
 */
(nic_state, mem_req): transmit_step2(nic_state: nic)
    nat: bit_shift := (nic.tx_p.transmit_read_buffer_descriptor_bytes % 4) * 8
    word8: bd_byte := nic.regs.CPPI_RAM(nic.tx_p.transmit_current_bd - 0x4A10_2000 +

    nic.tx_p.transmit_read_buffer_descriptor_bytes)

    if nic.tx_p.transmit_read_buffer_descriptor_bytes < 4 then
        nic.tx_p.transmit_current_bd_value_word0 :|= bd_byte << bit_shift
    else if nic.tx_p.transmit_read_buffer_descriptor_bytes < 8 then
        nic.tx_p.transmit_current_bd_value_word1 :|= bd_byte << bit_shift
    else if nic.tx_p.transmit_read_buffer_descriptor_bytes < 12 then
        nic.tx_p.transmit_current_bd_value_word2 :|= bd_byte << bit_shift
    else if transmit_read_buffer_descriptor_bytes < 16 then
        nic.tx_p.transmit_current_bd_value_word3 :|= bd_byte << bit_shift
    else
        //This function should not be applied if there are no more buffer
        //descriptor bytes to read. That is, a model bug.
        nic.dead_state := true
        return (nic, no_mem_req)

    //One additional byte has been read of the current buffer descriptor.
    nic.tx_p.transmit_read_buffer_descriptor_bytes :+= 1

    //Step 3 shall perform the next NIC transition if all bytes have been read.
    if nic.tx_p.transmit_read_buffer_descriptor_bytes ≥ 16 then
        nic.tx_p.transmit_step := 3

    return (nic, no_mem_req)

/*
 *  Checks that the current buffer descriptor is correctly initialized. If not,
 *  then a dead state is entered according to the following checks on the
 *  current buffer descriptor:
 *  -A SOP buffer descriptor is expected but the SOP bit is cleared.
 *  -A non-SOP buffer descriptor is expected but the SOP bit is set.
 *  -It is an expected SOP buffer descriptor but, either the ownership bit is
 *   cleared or the buffer offset field is greater than or equal to the buffer
 *   length field.
 *  -The buffer length field is zero or the EOQ bit is set.
 *
 *  If all checks are passed, then some variables are initialized such that:
 *  -The SOP buffer descriptor can be accessed by later step functions.
 *  -The packet length field of a SOP buffer descriptor can be checked to be
 *   correct.
 *  -It is known where to start fetch bytes from of the associated data buffer.
 *
 *  Finally, step function 4 is applied.
 */
(nic_state, mem_req): transmit_step3(nic_state: nic)
    //Retrieves the buffer pointer, offset and length fields
    word32: transmit_buffer_pointer := nic.tx_p.transmit_current_bd_value_word1
    word32: transmit_buffer_offset := nic.tx_p.transmit_current_bd_value_word2[31:16]
    nic.tx_p.transmit_buffer_length := nic.tx_p.transmit_current_bd_value_word2[15:0]

    //Checks if it is a SOP descriptor and in that case that everyting is setup
    //correctly.
    //Expects a SOP but it isn't.
    if nic.tx_p.transmit_expect_sop ∧  nic.tx_p.transmit_current_bd_value_word3[31] = 0 then
        nic.dead_state := true
        return (nic, no_mem_req)
    //Does not expect a sop but it is.
    else if ¬nic.tx_p.transmit_expect_sop ∧ nic.tx_p.transmit_current_bd_value_word3[31] = 1 then
        nic.dead_state := true
        return (nic, no_mem_req)
    //Expects SOP and it is.
    else if nic.tx_p.transmit_expect_sop ∧ nic.tx_p.transmit_current_bd_value_word3[31] = 1 then
        //Ownership bit must be set in SOP buffer descriptor.
        if nic.tx_p.transmit_current_bd_value_word3[29] = 0 then
            nic.dead_state := true
            return (nic, no_mem_req)
        //Buffer offset must be smaller than the buffer length. This is only
        //checked on SOP buffer descriptors since the buffer offset is only
        //valid in them. From the NIC specification: "The host sets the
        //buffer_offset value (which may be zero to the buffer length minus 1).
        //Valid only on sop."
        else if transmit_buffer_offset ≥ nic.tx_p.transmit_buffer_length then
            nic.dead_state := true
            return (nic, no_mem_req)

        //Records the address of this SOP buffer descriptor, since it is needed
        //later to clear the ownership bit.
        nic.tx_p.transmit_sop_bd_physical_address := nic.tx_p.transmit_current_bd

        //The SOP buffer descriptor is now consumed. It is set again when an
        //EOP buffer descriptor is encountered or when TX0_HDP is set.
        nic.tx_p.transmit_expect_sop := false

        //Retrieves the packet length field which is needed for later use to
        //check that the packet length field is of correct length: Equal to the
        //sum of the buffer length fields of all buffer descriptors of the
        //current frame under transmission. Therefore is
        //transmit_sum_buffer_length initialized to the buffer length field of
        //the first buffer descriptor of the current frame to transmit.
        nic.tx_p.transmit_sop_packet_length := nic.tx_p.transmit_current_bd_value_word3[10:0]
        nic.tx_p.transmit_sum_buffer_length := nic.tx_p.transmit_buffer_length
    //If the current buffer descriptor is not a SOP and a SOP is not expected,
        //then the buffer descriptor's buffer length field is used to increment
        //transmit_sum_buffer_length.
        else
                nic.tx_p.transmit_sum_buffer_length :+= nic.tx_p.transmit_buffer_length

        //The buffer length must be greater than zero and the EOQ bit cleared.
        if nic.tx_p.transmit_buffer_length = 0 ∨ nic.tx_p.transmit_current_bd_value_word3[28] = 1 then
                nic.dead_state := true
                return (nic, no_mem_req)

        //Sets the address of the next byte to be fetched from memory.
        transmit_next_buffer_byte_address := transmit_buffer_pointer
        //If this is a SOP then the buffer offset must be added to start fetching
        //the frame bytes from the right position. The offset field is only valid
        //in SOP buffere descriptors.
        if transmit_current_bd_value_word3[31] = 1 then
                transmit_next_buffer_byte_address :+= transmit_buffer_offset

        //Since this step function is considered to not do any hardware work, step
        //function four is applied.
        nic.tx_p.transmit_step := 4
        return transmit_step4(nic)

/*
 *      Issues memory read requests of one byte for each application of this
 *      function from the associated data buffer of the current buffer descriptor.
 *      The NIC scheduler can only apply this step function if the variable
 *      memory_request is falsified. That occurs by the framework when it applies
 *      memory_request_byte_reply and memory_request_byte_reply considers the
 *      memory request reply to be correct. When all memory requests have been
 *      issued, step function five will be applied the next time the transmission
 *      process is to make a transition.
 *
 *      If the buffer descriptor wraps around the memory border, or access memory
 *      outside RAM, then a dead state is entered. The RAM addresses on BeagleBone
 *      Black is [0x8000_0000, 0x8000_0000 + 0x2000_0000), 512 MB.
 */
(nic_state, mem_req): transmit_step4(nic_state: nic)
        //It is a model bug if this step function is applied when it should not be.
        if nic.tx_p.memory_request = true ∨ nic.tx_p.transmit_buffer_length = 0 then
                nic.dead_state := true
                return (nic, no_mem_req)
        else
                //Increments the address for next access.
                nic.tx_p.transmit_next_buffer_byte_address :+= 1
                //A memory request is pending.
                nic.tx_p.memory_request := true
                //One byte request less to make.
                nic.tx_p.transmit_buffer_length :-= 1

                //All memory requests have been issued.
                if nic.tx_p.transmit_buffer_length = 0 then
                        nic.tx_p.transmit_step := 5
                //Checks if the next memory request address wraps around or addresses
                //memory outside RAM.
                else if nic.tx_p.transmit_next_buffer_byte_address = 0 ∨
                                nic.tx_p.transmit_next_buffer_byte_address - 1 < 0x8000_0000 ∨
                                nic.tx_p.transmit_next_buffer_byte_address - 1 ≥ 0xA000_0000 then
                        nic.dead_state := true
                        return (nic, no_mem_req)

                //Returns a tuple that represents a memory read request.
                //mem_req = (valid, read, address, value).
                return (nic, (true, true, nic.tx_p.transmit_next_buffer_byte_address - 1, 0))

/*
 *      The following function steps (step 5 to 8) configures bits in the the SOP
 *      and EOP buffer descriptors, if the currently processed buffer descriptor
 *      is of type EOP:
 *      1.      Sets the EOQ bit if it is the last buffer descriptor in the transmit
 *              queue.
 *      2.      Clears the Ownership bit in the SOP buffer descriptor. Also clears the
 *          TX0_HDP register if this is the last buffer descriptor in the transmit
 *          queue.
 *     3.   Writes the physical address of the EOP buffer descriptor of the
 *          transmitted frame to the TX0_CP register.
 */

/*
 *     transmit_step5 considers three cases:
 *     1.   If the current buffer descriptor is not of type EOP but is the last one
 *          of the current frame under transmission, then this is an error.
 *     2.   If the current buffer descriptor is not of type EOP and is not the last
 *          of the current frame under transmission, then the next buffer
 *          descriptor is processed by applying step function one with
 *          transmit_current_bd pointing to the next buffer descriptor.
 *     3.   Otherwise the current buffer descriptor is of type EOP and
 *          transmit_step6 is applied since no hardware work was done.
 */
(nic_state, mem_req): transmit_step5(nic_state: nic)
    if nic.tx_p.transmit_current_bd_value_word3[30] = 0 ∧ nic.tx_p.transmit_current_bd_value_word0 = 0 then
        nic.dead_state := true
        return (nic, no_mem_req)
    else if nic.tx_p.transmit_current_bd_value_word3[30] = 0 ∧ nic.tx_p.transmit_current_bd_value_word0 ≠ 0 then
        nic.tx_p.transmit_current_bd := nic.tx_p.transmit_current_bd_value_word0
        nic.tx_p.transmit_step := 1
        return transmit_step1(nic)
    else
        return transmit_step6(nic)

/*
 *     A misqueue condition is detected when the following three conditions are
 *     true:
 *     -SOP has cleared ownership bit
 *     -EOP has set EOQ bit
 *     -EOP has non-zero Next Descriptor Pointer field.
 *
 *     Cnsider the following additional statements derived from the specification:
 *     -A misqueue condition is corrected by writing TX0_HDP with the new buffer
 *     descriptor.
 *     -Writing TX0_HDP when it is not zero is an error.
 *     -TX0_HDP is zero after transmission is complete.
 *     -TX0_HDP is written to initiate new transmission.
 *
 *     These five conditions imply that when a misqueue condition occurs, TX0_HDP
 *     is zero. They also imply that when TX0_HDP is zero, then all buffer descriptors
 *     are released by the NIC and the NIC will not modify them any     more.
 *
 *     That is, when the Ownership and EOQ bits are cleared and set, then TX0_HDP
 *     must be cleared. When TX0_HDP is cleared, the ownership and EOQ bits be
 *     cleared and set.
 *
 *     Since the ownership bit must be cleared after the EOQ bit is set. The EOQ
 *     bit is set first and then are the ownership bit and the TX0_HDP cleared
 *     atomically. This corresponds to steps 6 and 7.
 */

/*
 *     Sets the EOQ bit if this is the last buffer descriptor in the transmission
 *     queue. Also checks that the sum of the buffer length fields of the frame's
 *     buffer descriptors is not too big and that it is equal to the packet length
 *     field of the SOP buffer descriptor.
 */
(nic_state, mem_req): transmit_step6(nic_state: nic)
    //Checks errors first. If the sum of the buffer length fields is to big or
    //that sum is not equal to the packet length field of the SOP buffer
    //descriptor, at the time when it was read, then the NIC enters a dead
    //state.
    if nic.tx_p.transmit_sum_buffer_length ≥ 0x800 ∨
            nic.tx_p.transmit_sop_packet_length ≠ nic.tx_p.transmit_sum_buffer_length then
        nic.dead_state := true
        return (nic, no_mem_req)

    //This is EOP but next buffer descriptor shall be a SOP.
```

```
        nic.tx_p.transmit_expect_sop := true

        nic.tx_p.transmit_step := 7

        //Sets EOQ in last buffer descriptor.
        if nic.tx_p.transmit_current_bd_value_word0 = 0 then
                nic.regs.CPPI_RAM(nic.tx_p.transmit_current_bd - 0x4A10_2000 + 15)[4] := 1
                return (nic, no_mem_req)
        else
                return transmit_step7(nic)

/*
 *      Clears the ownership bit and the TX0_HDP register.
 */
(nic_state, mem_req): transmit_step7(nic_state: nic)
        //Clears ownership bit.
        nic.regs.CPPI_RAM(nic.tx_p.transmit_sop_bd_physical_address - 0x4A10_2000 + 15)[5] := 0

        //Clears TX0_HDP.
        if nic.tx_p.transmit_current_bd_value_word0 = 0 then
                nic.regs.TX0_HDP := 0

        //Stores the physical address of the EOP buffer descriptor used to write
        //TX0_CP.
        nic.tx_p.transmit_eop_bd_physical_address = nic.tx_p.transmit_current_bd

        //Advances the buffer descriptor pointer.
        nic.tx_p.transmit_current_bd := nic.tx_p.transmit_current_bd_value_word0

        nic.tx_p.transmit_step := 8
        return (nic, no_mem_req)

/*
 *      TX0_CP is set to the physical address of the currently transmitted frame's
 *      EOP buffer descriptor. A frame transmission completion interrupt is
 *      asserted non-deterministically since the interrupt related registers are
 *      not modeled. If there are no more frames to transmit or if there is are
 *      pending initialization or transmission teardown processes, then the
 *      transmission process gets terminated. Otherwise there are no pending
 *      initialization or transmission teardown processes and there are more frames
 *      to transmit and the transmission process start from beginning again.
 */
(nic_state, mem_req): transmit_step8(nic_state: nic)
        nic.regs.TX0_CP := nic.tx_p.transmit_eop_bd_physical_address
        if choice_non-deterministically({false, true}) then
                nic.tx_p.interrupt := true
                nic.interrupt := true

        if nic.tx_p.transmit_current_bd = 0 ∨ nic.tx_td_p.transmit_teardown_step = 1 ∨ nic.init_p.init_step = 1 then
                nic.tx_p.transmit_step := 0
        else
                nic.tx_p.transmit_step := 1

        return (nic, no_mem_req)

/*
 *      @memory_reply: The reply of the earlier issued memory request made in step
 *      four.
 *
 *      This function is applied by the device model framework to reply to an
 *      earlier memory read request issued by step function four of the
 *      transmission process. If the given "memory request" matches the expected
 *      reply of the memory request issued by step function four, then that memory
 *      request is considered satisfied and memory_request is falsified. Otherwise
 *      a dead state is entered.
 *
 *      The transmission process should be at step four (transmit_step = 4) when
 *      this step function is applied, because that is where the memory requests
 *      are issued.
 */
nic_state: memory_request_byte_reply(nic_state: nic, mem_req: memory_reply)
```

```
    if nic.tx_p.memory_request ∧ memory_reply.valid ∧ memory_reply.read ∧
            memory_reply.address = nic.tx_p.transmit_next_buffer_byte_address - 1 then
        nic.tx_p.memory_request := false
    else
        nic.dead_state := true

    return nic
```

C.9 Transmission Teardown Automaton

```
/*
 *   The following functions represents the operations of the transmission
 *   teardown process, which terminates the transmission process of the NIC.
 *   This transmission teardown process of the NIC gets activated when
 *   TX_TEARDOWN is written to zero. The transmission teardown process can start
 *   as soon as the transmission process has finished the processing of the
 *   frame currently being transmitted.
 *
 *   When the teardown process starts, transmit_current_bd points to the first
 *   buffer descriptor that is unused by the transmission process after the
 *   transmission process has terminated. If there is none, then
 *   transmit_current_bd is zero. This potentially existing buffer descriptor
 *   gets some of its fields modified by the this transmission teardown process.
 *   This is done by the following step functions:
 *   1.  Non-deterministically sets the EOQ bit in the first unused buffer
 *       descriptor after the terminated transmission process, if there is one.
 *   2.  Sets the teardown bit in the first unused buffer descriptor, if it
 *       there is one.
 *   3.  Clears TX0_HDP and the ownership bit in the first unused buffer
 *       descriptor, if there is one.
 *   4.  Writes the teardown acknowledgement code 0xFFFFFFFC in the TX0_CP
 *       register.
 */

/*
 *   Function: Called by the NIC scheduler when the NIC transmission teardown
 *   process is to make its next transition.
 */
(nic_state, mem_req): transmit_teardown(nic_state: nic)
    if nic.tx_td_p.transmit_teardown_step = 1 then
        return transmit_teardown_step1(nic)
    else if nic.tx_td_p.transmit_teardown_step = 2 then
        return transmit_teardown_step2(nic)
    else if nic.tx_td_p.transmit_teardown_step = 3 then
        return transmit_teardown_step3(nic)
    else if nic.tx_td_p.transmit_teardown_step = 4 then
        return transmit_teardown_step4(nic)
    else
        nic.dead_state := true
        return (nic, no_mem_req)

/*
 *   Sets the EOQ bit non-deterministically since it is not specified by the NIC
 *   specification that this operation is actually performed for teardowns. By
 *   tests on the real hardware, it can be seen that the bit is set. As a
 *   compromise between actual implementation and the specification, it is set
 *   non-deterministically.
 */
(nic_state, mem_req): transmit_teardown_step1(nic_state: nic)
    nic.tx_td_p.transmit_teardown_step := 2
    if choice_non-deterministically(bool) ∧ nic.tx_p.transmit_current_bd ≠ 0 then
        nic.regs.CPPI_RAM(nic.tx_p.transmit_current_bd - 0x4A10_2000 + 15)[4] := 1
        return (nic, no_mem_req)
    else
        return transmit_teardown_step2()

/*
 *   Sets the teardown bit on the potentially first unused buffer descriptor in
 *   the transmission queue. If no such buffer descriptor exists, step 3 is
 *   applied since then no hardware work is performed by this function.
 */
```

```
(nic_state, mem_req): transmit_teardown_step2(nic_state: nic)
    nic.tx_td_p.transmit_teardown_step := 3
    if nic.tx_p.transmit_current_bd ≠ 0 then
        nic.regs.CPPI_RAM(nic.tx_p.transmit_current_bd - 0x4A10_2000 + 15)[3] := 1
        return (nic, no_mem_req)
    else
        return transmit_teardown_step3()

/*
 *  Clears TX0_HDP and the ownership bit of the potentially existing first
 *  unused buffer descriptor. This is done atomically to be consistent with the
 *  transmission process which also performs these two operations atomically.
 *  This indicates to software that the teardown bit is valid since this bit is
 *  cleared when the NIC is done with a buffer descriptor.
 */
(nic_state, mem_req): transmit_teardown_step3(nic_state: nic)
    nic.tx_td_p.transmit_teardown_step := 4
    nic.regs.TX0_HDP := 0

    if nic.tx_p.transmit_current_bd ≠ 0 then
        nic.regs.CPPI_RAM(nic.tx_p.transmit_current_bd - 0x4A10_2000 + 15)[5] := 0
        nic.tx_p.transmit_current_bd := 0
    else
        nic.tx_p.transmit_current_bd := 0
        return transmit_teardown_step4()

    return (nic, no_mem_req)

/*
 *  Sets TX0_CP to the teardown completion code and transmit_teardown_step to
 *  zero to indicate that the teardown process has terminated.
 *
 *  A reception teardown interrupt is asserted non-deterministically since the
 *  interrupt related registers are not modeled.
 */
(nic_state, mem_req): transmit_teardown_step4(nic_state: nic)
    nic.regs.TX0_CP := 0xFFFFFFFC
    nic.tx_td_p.transmit_teardown_step := 0
    if choice_non-deterministically(bool) then
        nic.tx_p.interrupt := true
        nic.interrupt := true
    return (nic, no_mem_req)
```

C.10 NIC Registers Related to Interrupts

This section describes which NIC registers that need to be modeled in order to accurately specify how the NIC asserts the interrupts that are used by Linux. The NIC can assert four different interrupts:

- Frame transmission/reception completion interrupts: Is asserted when the NIC writes TX0_CP/RX0_CP with the physical address of the EOP buffer descriptor of the most recently transmitted/received frame, and is deasserted when the CPU writes TX0_CP/RX0_CP with the value it currently contains.
- Receive threshold interrupt: Is asserted when the number of free buffer descriptors in the receive queue channel zero gets below a certain (configurable) value, and is deasserted when the CPU writes the RX0_CP register with the content of that register.
- Miscellaneous interrupts: Is asserted when events occur that are related to timing, statistics, errors and the physical link.

This subsection explains how the first two interrupts can be added to the NIC model. These are the two interrupts that are used by the NIC driver in Linux 3.10.

To accurately model frame transmission completion interrupts, the following registers must be added to the model:

- C0_TX_EN: Bit b is set to one by the CPU to enable transmission DMA channel b completion interrupts.
- C0_TX_STAT: Read only register with bit b set by the NIC if interrupts from transmission DMA channel b are enabled and channel b asserts an interrupt.
- TX_INTMASK_SET: Bit b is set to one by the CPU to enable the NIC DMA engine to generate interrupts for transmission DMA channel b.
- TX_INTMASK_CLEAR: Bit b is set to one by the CPU to disable the NIC DMA engine to generate interrupts for transmission DMA channel b. It is unspecified what happens to bit b in this and the TX_INTMASK_SET register when bit b is set in these two registers. They could be registers that are always read as zero.
- TX_INTSTAT_RAW: Read only register with bit b set to one if the NIC DMA engine asserts an interrupt for transmission channel b, even though it is disabled (by setting bit b in TX_INTMASK_CLEAR).
- TX_INTSTAT_MASKED: Read only register with bit b set to one if the NIC DMA engine asserts an interrupt for transmission channel b and that interrupt is enabled (by writing a one to bit b of the TX_INTMASK_SET register).

The corresponding registers for the frame reception completion interrupts behave identically as for the frame transmission completion interrupts: C0_RX_EN, C0_RX_STAT, RX_INTMASK_SET, RX_INTMASK_CLEAR, RX_INTSTAT_RAW, and RX_INTSTAT_MASKED.

The end of interrupt register, CPDMA_EOI_VECTOR, has five relevant bits. The software writes this register to two/one immediately after it has acknowledged a frame transmission/reception interrupt by writing the TX0_CP/RX0_CP register.

Appendix D Sub-Execution Trace in Real Model

This section illustrates what the execution traces of the real model look like, and how the traces are constructed and their states modified by applying the transition rules that defines the real model. It will also be shown how the non-determinism of some of the NIC transition rules is handled when applying such transition rules, and how the method of applying transition rules makes it unnecessary to define the non-deterministic scheduler of the device model framework.

Assume that the state (cpu_0, $memory_0$, nic_0) has the following properties:

- The MMU maps the program counter ($cpu_0.uregs.r15$) to physical address 0x0.
- The instruction in memory at 0x0 is 'ADD R0, R1, R2', which adds the numbers in $cpu_0.uregs.r1$ and $cpu_0.uregs.r2$ and stores the result in $cpu_0.uregs.r0$ (0x0 is not a part of RAM but chosen for simplicity):

  $$memory_0(\text{0x3}) :: memory_0(\text{0x2}) :: memory_0(\text{0x1}) :: memory_0(\text{0x0}) = \text{'ADD R0, R1, R2'}.$$

- The first three user registers have the following content:

  $$cpu_0.uregs.r0 = 0 \land cpu_0.uregs.r1 = 1 \land cpu_0.uregs.r2 = 2.$$

- The NIC is in a state such that it can perform the last step of the transmission teardown process with $TX0\_CP$ set to 0x4A102800:

  $$\neg nic_0.dead\_state \land$$
  $$nic_0.tx\_td\_p.transmit\_teardown\_step = 4 \land nic_0.tx\_p.transmit\_step = 0 \land$$
  $$nic_0.regs.TX0\_CP = \text{0x4A10\_2800}.$$

An execution trace with two transitions starting from (cpu_0, $memory_0$, nic_0) can be constructed as follows, where the first transition is performed by the CPU and the second by the NIC. The trace is constructed by applying the transition rules that defines the real model. In order to do that, their premises must be true. When the premise of a transition rule is true, that transition rule can be used to generate a transition. If several transition rules have their premises true simultaneously, then any one of them can be applied leading to several possible transitions from a given state. This is the reason why the transition rules do not need to consider the non-deterministic system scheduler of the device model framework.

The first transition is (cpu_0, $memory_0$, nic_0) $\rightarrow_{CPU}$ (cpu_1, $memory_1$, nic_1). It is generated by the CPU transition rule that only affects the CPU and the memory:

$$\frac{(cpu, memory) \rightarrow_{ARMv7} (cpu', memory') \land \neg nic\_access(cpu, memory)}{(cpu, memory, nic) \rightarrow_{\{CPU, EXC, EXC\_RET\}} (cpu', memory', nic)}.$$

This transition rule is instantiated as:

$$\frac{(\mathit{cpu}_0, \mathit{memory}_0) \rightarrow_{ARMv7} (\mathit{cpu}_1, \mathit{memory}_1) \land \neg \mathit{nic\_access}(\mathit{cpu}_0, \mathit{memory}_0)}{(\mathit{cpu}_0, \mathit{memory}_0, \mathit{nic}_0) \rightarrow_{CPU} (\mathit{cpu}_1, \mathit{memory}_1, \mathit{nic}_1)}.$$

According to the ARMv7 specification, the CPU will execute the ADD instruction, and not access the memory (except for instruction fetch) or the NIC. This means that $\neg \mathit{nic\_access}(\mathit{cpu}_0, \mathit{memory}_0) = \mathit{true}$ and $\mathit{cpu}_1.\mathit{uregs}.\mathit{r0} = 3$. Since memory_0 and nic_0 are not modified, $\mathit{memory}_0 = \mathit{memory}_1$ and $\mathit{nic}_0 = \mathit{nic}_1$.

The second transition is $(\mathit{cpu}_1, \mathit{memory}_1, \mathit{nic}_1) \rightarrow_{NIC} (\mathit{cpu}_2, \mathit{memory}_2, \mathit{nic}_2)$. It is generated by the transition rule for autonomous transitions without memory requests:

$$\frac{\mathit{nic} \rightarrow_{\mathit{nic}(\mathit{false}, \mathit{read}, \mathit{pa}, \mathit{val})} \mathit{nic}'}{(\mathit{cpu}, \mathit{memory}, \mathit{nic}) \rightarrow_{NIC} (\mathit{cpu}, \mathit{memory}, \mathit{nic}')}.$$

This transition rule is instantiated as:

$$\frac{\mathit{nic}_1 \rightarrow_{\mathit{nic}(\mathit{false}, \mathit{false}, 0, 0)} \mathit{nic}_2}{(\mathit{cpu}_1, \mathit{memory}_1, \mathit{nic}_1) \rightarrow_{NIC} (\mathit{cpu}_2, \mathit{memory}_2, \mathit{nic}_2)},$$

where *read*, *pa* and *val* has been assigned the values *false*, 0 and 0, respectively. The transition of the premise in this transition rule is generated by a NIC model transition rule. The transition rule of the NIC model that generates such a transition is:

$$\frac{(\mathit{nic}', (\mathit{valid}, \mathit{read}, \mathit{pa}, \mathit{val}), \mathit{int}) = \mathit{nic\_scheduler}(\mathit{nic})}{\mathit{nic} \rightarrow_{\mathit{nic}(\mathit{valid}, \mathit{read}, \mathit{pa}, \mathit{val})} \mathit{nic}'}.$$

This transition rule is instantiated as:

$$\frac{(\mathit{nic}_2, (\mathit{false}, \mathit{false}, 0, 0), \mathit{true}) = \mathit{nic\_scheduler}(\mathit{nic}_1)}{\mathit{nic}_1 \rightarrow_{\mathit{nic}(\mathit{false}, \mathit{false}, 0, 0)} \mathit{nic}_2}.$$

where *int* has been assigned the value *true*. Now it must be established that $\mathit{nic\_scheduer}(\mathit{nic}_1)$ can return the value $(\mathit{nic}_2, (\mathit{false}, \mathit{false}, 0, 0), \mathit{true})$.

By investigating the pseudocode of *nic_scheduler*, it can be seen that the element *transmit_teardown* is added to *P* since

$$\neg \mathit{nic}_1.\mathit{dead\_state} \land \\ \mathit{nic}_1.\mathit{tx\_td\_p}.\mathit{transmit\_teardown\_step} = 4 \land \mathit{nic}_1.\mathit{tx\_p}.\mathit{transmit\_step} = 0$$

holds. Since *nic_scheduler* non-deterministically selects an element in *P*, an arbitrary element in *P* can be chosen when applying this rule and therefore is the element *transmit_teardown* a valid selection. *nic_scheduer* then applies *transmit_teardown* which in turn applies *transmit_teardown_step4*. *transmit_teardown_step4* sets *TX0_CP* to 0xFFFFFFFC, *transmit_teardown_step* to 0, non-deterministically sets the interrupt flag and then returns the modified NIC state along with an empty memory request: $(\mathit{nic}_2, \mathit{no\_mem\_req})$, where *no_mem_req* is equal to (*false*, *false*, 0, 0). Since the interrupt flag is set non-deterministically, can it be chosen to be set to *true*. *nic_scheduler* can therefore return

$$(nic_2, (false, false, 0, 0), true).$$

This is consistent with the premise of the NIC transition rule above, which therefore generates the transition $nic_1 \rightarrow_{nic(false, false, 0, 0)} nic_2$. This also makes the premise true for the transition rule for autonomous transitions without memory requests of the real model, which therefore generates the transition

$$(cpu_1, memory_1, nic_1) \rightarrow_{NIC} (cpu_2, memory_2, nic_2),$$

where $cpu_1 = cpu_2 \land memory_1 = memory_2$ since the CPU and the memory are not accessed by this NIC transition.

The final trace is therefore

$$(cpu_0, memory_0, nic_0) \rightarrow_{CPU} (cpu_1, memory_1, nic_1) \rightarrow_{NIC} (cpu_2, memory_2, nic_2).$$

Since the two transition rules depend on and manipulate disjunct state components can they be reordered without leading to the premises of their transition rules becoming false. This gives another trace:

$$(cpu_0, memory_0, nic_0) \rightarrow_{NIC} (cpu_3, memory_3, nic_3) \rightarrow_{CPU} (cpu_2, memory_2, nic_2),$$

where the intermediate state $(cpu_3, memory_3, nic_3)$ is different from $(cpu_1, memory_1, nic_1)$, but where the start and end states are unmodified. Some of the state components of the final state $(cpu_2, memory_2, nic_2)$ has the following values:

- User register zero (except for the program counter) has been modified:

$$cpu_0.uregs.r0 = 3 \land cpu_0.uregs.r1 = 1 \land cpu_0.uregs.r2 = 2.$$

- *transmit_teardown_step* and *TX0_CP* has been modified:

$$\neg nic_0.dead\_state \land$$
$$nic_0.tx\_td\_p.transmit\_teardown\_step = 0 \land nic_0.tx\_p.transmit\_step = 0 \land$$
$$nic_0.regs.TX0\_CP = 0\text{xFFFFFFFC}.$$

Appendix E Definition of SEC

This appendix formally defines the security invariant *SEC*, which is defined as the conjunction of *CPU_MEMORY* and *NIC*. First is some notation described together with some definitions of functions and predicates that are used to define *SEC*. Second is *CPU_MEMORY* defined and third *NIC*. The last section lists which state components each predicate of *SEC* depends on.

E.1 Notation and Definitions

As in Section 6.1, functions that are defined using classic mathematical notation are defined by means of the symbol '$\overset{\text{def}}{=}$', and the other definitions are according to the pseudocode syntax described in Appendix A. '[' and ']' are used as regular parentheses. Also, NIC transitions $s \rightarrow t$ that correspond to a memory access at physical address *pa* reading or writing the value *v* have the labels *nic_memory_read*(*pa,v*) or *nic_memory_write*(*pa,v*), respectively.

Some definitions that are used in several predicates:

- *CPPI_RAM* = 0x4A102000: First physical address of CPPI_RAM.
- word32: *cppi_ram_word*(ideal_state: *i*, word32: *bd_ptr*) returns the 32-bit word stored in CPPI_RAM at the physical address *bd_ptr* in the state *i*:

```
word32: cppi_ram_word(ideal_state: i, word32: bd_ptr):
    return i.nic.regs.CPPI_RAM(bd_ptr – CPPI_RAM + 3) ::
        i.nic.regs.CPPI_RAM(bd_ptr – CPPI_RAM + 2) ::
        i.nic.regs.CPPI_RAM(bd_ptr – CPPI_RAM + 1) ::
        i.nic.regs.CPPI_RAM(bd_ptr – CPPI_RAM).
```

- {word32}: *queue*(ideal_state: *i*, word32: *bd_ptr*) returns the set of physical addresses of all buffer descriptors in the queue pointed to by the physical address *bd_ptr*:

```
{word32}: queue(ideal_state: i, word32: bd_ptr):
    if bd_ptr = 0 then
        return {}
    else
        return {bd_ptr} ∪ queue(i, cppi_ram_word(i, bd_ptr)).
```

- {word32}: *bd_addresses*(word32: *bd_ptr*) returns the set of physical byte addresses in CPPI_RAM that the buffer descriptor at the physical address *bd_ptr* uses.

  {word32}: *bd_addresses*(word32: *bd_ptr*) $\overset{\text{def}}{=}$ $\{a \mid bd\_ptr \leq a \leq bd\_ptr + \text{0xF}\}$.

- (word32, word32): *first_last_tx_block_index*(ideal_state: *i*, word32: *bd_ptr*) returns the block indexes of the first and last physical addresses of the memory region that the transmission buffer descriptor at physical address *bd_ptr* specifies:

```
(word32, word32): first_last_tx_block_index(ideal_state: i,
                                            word32: bd_ptr):
    word32: f := cppi_ram_word(i, bd_ptr + 4)
    word32: l := cppi_ram_word(i, bd_ptr + 4) +
                 cppi_ram_word(i, bd_ptr + 8)[15:0] – 1
    word32: offset:= cppi_ram_word(i, bd_ptr + 8)[31:16])
    if cppi_ram_word(i, bd_ptr + 12)[31] = 0b1 then
        return ((f + offset)[31:12], (l + offset)[31:12])
    else
        return (f[31:12], l[31:12]).
```

- {word20}: *allocated_tx_blocks*(ideal_state: *i*, word32: *bd_ptr*) returns the block indexes of the blocks accessed by the transmission buffer descriptor at physical address *bd_ptr*.

  {word20}: *allocated_tx_blocks*(ideal_state: *i*, word32: *bd_ptr*) $\overset{\text{def}}{=}$
  {*bl* | ∃*f*, *l* ∈ word20. (*f*, *l*) = *first_last_tx_block_index*(*i*, *bd_ptr*) ∧
  bl ∈ [*f*, *l*]}.

- {word20}: *allocated_rx_blocks*(ideal_state: *i*, word32: *bd_ptr*) returns the block indexes of all memory blocks accessed by the receive buffer descriptor at *bd_ptr*, by reading *i.oracle.recv_bd_nr_blocks*. The offset field is not considered since *RX_BUFFER_OFFSET* is required to be zero. *x* >> *y* means shifting *x* *y* bits to the right and inserting zeros at the most significant bits.

  {word20}: *allocated_rx_blocks*(ideal_state: *i*, word32: *bd_ptr*) $\overset{\text{def}}{=}$
  [*cppi_ram_word*(*i*, *bd_ptr* + 4)[31:12],
  cppi_ram_word(*i*, *bd_ptr* + 4)[31:12] +
  (*i.oracle.recv_bd_nr_blocks*((*bd_ptr* – *CPPI_RAM*) >> 2) – 1)].

- *NIC_execution*(ideal_state: *i*) returns the set of all possible execution traces that start from the state *i* and where only NIC transitions are made by means of the transition rules used to define the ideal model, and the NIC is idle in the end state (meaning that successive autonomous transitions does not manipulate the NIC state).

- [(word20, bool, bool, bool)]: *PTL1*(ideal_state: *i*, word20: *pt*): Returns a list with entries that contain information about how each block is mapped by the block *pt*, by interpreting *pt* as a first-level page table. If *pt* is mapping a block *bl* with with read, write and execute access permissions as indicated by the boolean variables *rd*, *wt* and *ex*, then is the tuple (*bl*, *rd*, *wt*, *ex*) in the list *PTL1*(*i*, *pt*). Entries that are free or correspond to second-level pointers are not in the returned list.

- [(word20, bool, bool, bool)]: *PTL2*(ideal_state: *i*, word20: *pt*): As *PTL2* but interprets *pt* as a second-level page table.

E.2 Definition of CPU_MEMORY

CPU_MEMORY is defined as:

$$\mathit{CPU\_MEMORY}(\text{ideal\_state: } i) \overset{\text{def}}{=}$$
$$\mathit{WT\_EX\_REF}(i) \land \mathit{SOUND\_PT}(i) \land \mathit{CONST\_PT}(i) \land \mathit{SOUND\_MMU}(i) \land \mathit{IN\_LINUX}(i).$$

These predicates are described and defined in the following subsections.

E.2.1 WT_EX_REF

WT_EX_REF:

ρ_{wt} and ρ_{ex} is correct: $\rho_{wt}(bl)$ and $\rho_{ex}(bl)$ records the number of page table entries in L1 and L2 blocks that map bl as writable and executable, respectively.

This property allows security related decisions to be based on ρ_{wt} and ρ_{ex}.

In the formula below, the *j*th entry of a list *l* is accessed as $l[j]$.

$$\begin{aligned}
&\text{bool: } \mathit{WT\_EX\_REF}(\text{ideal\_state: } i) \overset{\text{def}}{=} \\
&\quad [\forall bl \in \text{word20}. \\
&\qquad i.oracle.\rho_{wt}(bl) = |\{(pt, j) \mid \exists pt \in \text{word20}. \\
&\qquad\qquad i.oracle.\tau(pt) = L1 \land (bl, rd, wt, ex) = PTL1(i, pt)[j] \land wt \lor \\
&\qquad\qquad i.oracle.\tau(pt) = L2 \land (bl, rd, wt, ex) = PTL2(i, pt)[j] \land wt\}|] \\
&\quad \land \\
&\quad [\forall bl \in \text{word20}. \\
&\qquad i.oracle.\rho_{ex}(bl) = |\{(pt, j) \mid \exists pt \in \text{word20}. \\
&\qquad\qquad i.oracle.\tau(pt) = L1 \land (bl, rd, wt, ex) = PTL1(i, pt)[j] \land ex \lor \\
&\qquad\qquad i.oracle.\tau(pt) = L2 \land (bl, rd, wt, ex) = PTL2(i, pt)[j] \land ex\}|]
\end{aligned}$$

Since *pt* identifies a unique page table and *j* a unique entry of that page table, each tuple represents exactly one unique page table entry.

E.2.2 SOUND_PT

SOUND_PT:

All L1 and L2 blocks have secure access permissions: Linux blocks are not both executable and writable, and if they are executable then their content is signed. In addition, all second-level entries of L1 blocks are referring to L2 blocks.

This property is used (with other predicates) to enforce Linux to only execute signed code.

Some help predicates are used to define *SOUND_PT*:

- bool: $\mathit{SOUND}_{W\oplus X}$(ideal_state: *i*, bool: *wt*, bool: *ex*, word20: *bl*) is used to state that a block mapped by a page table entry is not both writable and executable, and no other page table entry of any potential page table is in conflict with it (due to the inclusion of ρ_{wt} and ρ_{ex}).

  $$\begin{aligned}
  &\text{bool: } \mathit{SOUND}_{W\oplus X}(\text{ideal\_state: } i, \text{bool: } wt, \text{bool: } ex, \text{word20: } bl) \overset{\text{def}}{=} \\
  &\quad (ex \Rightarrow \neg wt \land i.oracle.\rho_{wt}(bl) = 0) \land (wt \Rightarrow \neg ex \land i.oracle.\rho_{ex}(bl) = 0).
  \end{aligned}$$

- bool: SOUND_S(ideal_state: *i*, bool: *ex*, bool: *bl*) is used to state that an executable block is signed according to the golden image:

$$
\begin{aligned}
&\text{bool: } \mathit{SOUND}_S(\text{ideal\_state: } i, \text{bool: } \mathit{ex}, \text{bool: } \mathit{bl}) \overset{\text{def}}{=}\\
&\quad \mathit{ex} \Rightarrow \mathit{i.oracle.sign}(\mathit{content}(i, \mathit{bl})) \in \mathit{i.oracle.GI}.
\end{aligned}
$$

- [word32]: *L2_ENTRY*(ideal_state: *i*, word20: *bl*) returns the list of all second-level page table link entries in the *L1* block *bl* in the state *i*.
- bool: *L2_ENTRY_L2_BL*(ideal_state: *i*, word20: *bl*) is true if and only if all second-level entries of an *L1* block *bl* point to *L2* blocks:

$$
\begin{aligned}
&\text{bool: } \mathit{L2\_ENTRY\_L2\_BL}(\text{ideal\_state: } i, \text{word20: } \mathit{bl}) \overset{\text{def}}{=}\\
&\quad \forall \mathit{pte} \in \mathit{L2\_ENTRY}(i, \mathit{bl}).\ \mathit{i.oracle}.\tau(\mathit{pte}[31{:}12]) = \mathit{L2}.
\end{aligned}
$$

$$
\begin{aligned}
&\text{bool: } \mathit{SOUND\_PT}(\text{ideal\_state: } i) \overset{\text{def}}{=}\\
&\quad \forall \mathit{bl} \in \text{word20}.\\
&\qquad [\mathit{i.oracle}.\tau(\mathit{bl}) = \mathit{L1}\\
&\qquad \Rightarrow\\
&\qquad [\forall(\mathit{pb}, \mathit{rd}, \mathit{wt}, \mathit{ex}) \in \mathit{PTL1}(i, \mathit{bl}).\\
&\qquad\qquad\qquad \mathit{SOUND}_{W\oplus X}(i, \mathit{wt}, \mathit{ex}, \mathit{pb}) \land \mathit{SOUND}_S(i, \mathit{ex}, \mathit{pb})] \land\\
&\qquad \mathit{L2\_ENTRY\_L2\_BL}(i, \mathit{bl})]]\\
&\qquad \land\\
&\qquad [\mathit{i.oracle}.\tau(\mathit{bl}) = \mathit{L2}\\
&\qquad \Rightarrow\\
&\qquad [\forall(\mathit{pb}, \mathit{rd}, \mathit{wt}, \mathit{ex}) \in \mathit{PTL2}(i, \mathit{bl}).\\
&\qquad\qquad \mathit{SOUND}_{W\oplus X}(i, \mathit{wt}, \mathit{ex}, \mathit{pb}) \land \mathit{SOUND}_S(i, \mathit{ex}, \mathit{pb})]].
\end{aligned}
$$

E.2.3 CONST_PT

CONST_PT:

Linux cannot write L1 or L2 blocks.

This property prohibits Linux from changing access permissions and access critical data like the golden image in the monitor.

$$
\begin{aligned}
&\text{bool: } \mathit{CONST\_PT}(\text{ideal\_state: } i) \overset{\text{def}}{=}\\
&\qquad \forall \mathit{bl} \in \text{word20}.\ \mathit{i.oracle}.\tau(\mathit{bl}) \in \{\mathit{L1}, \mathit{L2}\} \Rightarrow \mathit{i.oracle}.\rho_{wt}(\mathit{bl}) = 0.
\end{aligned}
$$

E.2.4 SOUND_MMU

SOUND_MMU:

The first-level page table used by the MMU uses an L1 block as its first-level page table. Also, the virtual addresses of the memory containing the code of the hypervisor and the monitor are correctly mapped to the expected physical addresses, and that physical memory contain their expected code.

Together with *SOUND_PT*, this property forces the MMU to actually apply the security policy, and that the hypervisor gets control when exceptions occur.

Definitions of constants and predicates:

- word32: *HYP_SIZE*/*MON_SIZE*: The number of bytes that constitutes the memory image of the code of the hypervisor and the monitor, respectively. Should be a multiple of 4096.

- word32: *HYP_VIRT*[*j*]/*MON_VIRT*[*j*] contains the *j*th virtual byte address that is expected to be mapped to the *j*th byte of the physical memory region allocated to the code of the hypervisor/monitor.
- word32: *HYP_PHYS*[*j*]/*MON_PHYS*[*j*] contains the *j*th physical byte address that is expected to contain the *j*th byte of the memory image that contains the code of the hypervisor/monitor. *HYP_VIRT*[j]/*MON_VIRT*[*j*] is expected to be mapped by the MMU to *HYP_PHYS*[*j*]/*MON_PHYS*[*j*].
- word32: *HYP_MEM*[*j*]/*MON_MEM*[*j*] contains the expected content of the *j*th byte of the memory image that contains the code of the hypervisor/monitor. *HYP_PHYS*[*j*]/*MON_PHYS*[*j*] is expected to contain the physical address of the memory location that contains the content of *HYP_MEM*[*j*]/*MON_MEM*[*j*].
- bool: *HVM_MAP*(ideal_state: *i*) requires that the MMU maps the virtual addresses of the code of the hypervisor and the monitor to their expected physical addresses:

  bool: $HVM\_MAP(\text{ideal\_state: } i) \overset{\text{def}}{=}$
  $[\forall 0 \leq j < HYP\_SIZE.$
  $mmu(i, PL1, HYP\_VIRT[j]\ \&\ {\sim}3, ex) = HYP\_PHYS[j]\ \&\ {\sim}03] \wedge$
  $[\forall 0 \leq j < MON\_SIZE.$
  $mmu(i, PL1, MON\_VIRT[j]\ \&\ {\sim}3, ex) = MON\_PHYS[j]\ \&\ {\sim}03].$

  The hypervisor and monitor addresses must be word aligned when given to the MMU which is the reason for zeroing the two least significant bits of the address.

bool: $SOUND\_MMU(\text{ideal\_state: } i) \overset{\text{def}}{=}$
$i.cpu.cp15.TTBR0[11{:}0]\ \&\ \text{0xFC0} = 0 \wedge$
$i.oracle.\tau(i.cpu.cp15.TTBR0[31{:}12]) = L1 \wedge$
$HVM\_MAP(i) \wedge$
$[\forall 0 \leq j < HYP\_SIZE.\ i.memory(HYP\_PHYS[j]) = HYP\_MEM[j]] \wedge$
$[\forall 0 \leq j < MON\_SIZE.\ i.memory(MON\_PHYS[j]) = MON\_MEM[j]].$

The first conjunct checks that some of the least significant bits of *TTBR0* are zero since it should specify an *L1* page table block. The reason for the 0xFC0 mask is that some bits of the *TTBR0* register have a special meaning that is not part of the specified *L1* page table block address. Some other bits of *TTBR0* are also not part of the *L1* page table block address but they are read as zero (which is the reason why the mask does not only contain ones in the 12 least significant bits).

E.2.5 LINUX

LINUX:

τ is correct:

- *Blocks typed as L1, L2 or D are part of Linux memory.*
- *Blocks typed as ⊥ are either unmapped by L1 and L2 blocks, and if they are mapped, then are they mapped as inaccessible to Linux.*

- *Blocks that correspond to NIC registers that affect which memory accesses the NIC does are typed as MN.*
- *The block that corresponds to NIC registers where no register affects which memory accesses the NIC does is typed as N.*
- *Blocks belonging to the memory of the hypervisor or the monitor are typed as* $\bot$.

Linux is executed in non-privileged mode with DACR[5:4] = 0b01.

The purpose of this predicate is to ensure that Linux executes in non-privileged mode with secure access permissions. It also allows the potential page tables to securely map hypervisor and monitor memory to ease the proof that the implementation follows the design.

Auxiliary predicates:

- bool: *NO_MAP*(ideal_state: *i*, word20: *bl*) is used to force the block *bl* to not be mapped by any entry in any *L1* and *L2* blocks.

  $$\begin{array}{l} \text{bool: } NO\_MAP(\text{ideal\_state: } i, \text{word20: } bl) \overset{\text{def}}{=} \\ \quad [\neg\exists pt \in \text{word20}, (pb, rd, wt, ex) \in PTL1(i, pt). \\ \qquad\qquad i.oracle.\tau(pt) = L1 \wedge pb = bl] \wedge \\ \quad [\neg\exists pt \in \text{word20}, (pb, rd, wt, ex) \in PTL2(i, pt). \\ \qquad\qquad i.oracle.\tau(pt) = L2 \wedge pb = bl]. \end{array}$$

- bool: *NO_AP*(ideal_state: *i*, word20: *bl*) is used to state that if the block *bl* is mapped by any *L1* or *L2* block page table entry, then it has no access permissions.

  $$\begin{array}{l} \text{bool: } NO\_AP(\text{ideal\_state: } i, \text{word20: } bl) \overset{\text{def}}{=} \\ \quad \forall pt \in \text{word20}. \\ \qquad [i.oracle.\tau(pt) = L1 \\ \qquad \Rightarrow \\ \qquad \forall(pb, rd, wt, ex) \in PTL1(i, pt).\, pb = bl \Rightarrow \neg rd \wedge \neg wt \wedge \neg ex] \wedge \\ \qquad [i.oracle.\tau(pt) = L2 \\ \qquad \Rightarrow \\ \qquad \forall(pb, rd, wt, ex) \in PTL2(i, pt).\, pb = bl \Rightarrow \neg rd \wedge \neg wt \wedge \neg ex]. \end{array}$$

$$\begin{array}{l} \text{bool: } IN\_LINUX(\text{ideal\_state: } i) \overset{\text{def}}{=} \\ \quad [\forall bl \in \text{word20}. \\ \qquad [i.oracle.\tau(bl) \in \{L1, L2, D\} \Rightarrow bl :: 0^{12} \in LINUX\_MEM] \wedge \\ \qquad [i.oracle.\tau(bl) = \bot \Rightarrow NO\_MAP(i, bl) \vee NO\_AP(i, bl)] \wedge \\ \qquad [bl \in \{\text{0x4A100, 0x4A102, 0x4A103}\} \Rightarrow i.oracle.\tau(bl) = MN] \wedge \\ \qquad [bl = \text{0x4A101} \Rightarrow i.oracle.\tau(bl) = N]] \wedge \\ \quad [\forall 0 \leq j < HYP\_SIZE.\ i.oracle.\tau(HYP\_PHYS[j][31{:}12]) = \bot] \wedge \\ \quad [\forall 0 \leq j < MON\_SIZE.\ i.oracle.\tau(MON\_PHYS[j][31{:}12]) = \bot] \wedge \\ \quad [Linux\_exec(i) \Rightarrow Linux\_state(i)]. \end{array}$$

E.3 Definition of NIC

NIC is defined as:

NIC(ideal_state: *i*) $\overset{\text{def}}{=}$
FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES(*i*) ∧ *NIC_BDS*(*i*) ∧
NO_BD_OVERLAPS(*i*) ∧ *NIC_DATA_NO_EXEC_CONF*(*i*) ∧
NIC_READ_ONLY(*i*) ∧ *CANNOT_DIE*(*i*) ∧ *TD_STOP_NIC*(*i*) ∧
RECV_BD_REF(*i*) ∧ *INIT_TD_IDLE*(*i*) ∧
RX_BUFFER_OFFSET_DMACONTROL_ZERO(*i*) ∧ *ACTIVE_CPPI_RAM*(*i*).

Each of these predicates are presented in the following subsections by an intuitive meaning, their purposes and their formal definition.

E.3.1 FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES

FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES:

The transmit and receive queues pointed to by tx0_active_queue and rx0_active_queue are: completely located in CPPI_RAM, word aligned and of finite length.

The main reason for including this predicate is to enable computation of other predicates using their formal definition since some of those predicates would be undefined for cyclic queues or when buffer descriptor addresses are outside *CPPI_RAM*.

Necessary auxiliary predicates is:

- bool: *FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE*(ideal_state: *i*, word32: *bd_ptr*, {word32}: *visited*)

  returns true if and only if the queue pointed to by the physical address *bd_ptr* is of finite length, word aligned and completely located in *CPPI_RAM*.

```
bool: FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE(ideal_state: i,
                                word32: bd_ptr, {word32}: visited):
    if $\mathit{bd\_ptr} = 0$ then
        return true
    else if $\mathit{bd\_ptr} < \text{0x4A102000} \lor \text{0x4A104000} - \text{0x10} < \mathit{bd\_ptr} \lor$
                $\mathit{bd\_ptr}[1{:}0] \neq 0$ then
        return false
    else if $\mathit{bd\_ptr} \in \text{visited}$ then
        return false
    else
        word32: next_bd_ptr := cppi_ram_word(i, bd_ptr)
        return FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE(i,
                                next_bd_ptr, $\{\mathit{bd\_ptr}\} \cup \mathit{visited}$).
```

The definition of *FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES* follows.

```
bool: FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES(ideal_state: i) $\overset{\text{def}}{=}$
        FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE(i,
                                i.oracle.tx0_active_queue, $\varnothing$) ∧
        FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE(i,
                                i.oracle.rx0_active_queue, $\varnothing$).
```

E.3.2 NIC_BDS

NIC_BDS:

All buffer descriptors that can potentially be in use by the transmission and reception processes of the NIC are in the queues pointed to by tx0_active_queue and rx0_active_queue, respectively, and for reception also requires correctness of recv_bd_nr_blocks. These variables point to either:

- *A released SOP buffer descriptor that precedes buffer descriptors that are in use by the NIC, if the NIC has not finished the processing of the queue.*
- *The first non-released SOP buffer descriptor that will be used to transmit/store the first part of the next transmitted/received frame.*
- *Nothing: Contains zero if the transmission/reception process has finished its memory transfers.*

For each buffer descriptor in the queue pointed to by rx0_active_queue, recv_bd_nr_blocks records the number of blocks that buffer descriptor can potentially access.

NIC_BDS allows the oracle to rely on *tx0_active_queue*, *rx0_active_queue* and *recv_bd_nr_blocks* when deciding whether a NIC register write request is secure or not.

The definition of *NIC_BDS* is complex and requires several auxiliary functions and predicates:

- word32: *next_sop*(ideal_state: *i*, word32: *bd_ptr*) returns the first SOP buffer descriptor in the queue pointed to by *bd_ptr*.

  ```
  word32: next_sop(ideal_state: i, word32: bd_ptr):
      if cppi_ram_word(i, bd_ptr + 12)[31] = 1 then
          return bd_ptr
      else
          return next_sop(i, cppi_ram_word(i, bd_ptr)).
  ```

- word32: *first_non_released_sop*(ideal_state: *i*, word32: *bd_ptr*) returns the first SOP buffer descriptor that is not released in the queue pointed to by *bd_ptr*.

  ```
  word32: first_non_released_sop(ideal_state: i, word32: bd_ptr):
      if cppi_ram_word(i, bd_ptr + 12)[29] = 1 then
          return bd_ptr
      else if cppi_ram_word(i, bd_ptr + 12)[30] = 1 then
          return first_non_released_sop(i, cppi_ram_word(i, bd_ptr))
      else
          bd_ptr := next_sop(i, cppi_ram_word(i, bd_ptr))
          return first_non_released_sop(i, bd_ptr).
  ```

- bool: *BD_EQ*(ideal_state: *i1*, ideal_state: *i2*, word32: *bd_ptr*) returns true if and only if the buffer descriptors at address *bd_ptr* are equal in the states *i1* and *i2*.

bool: *BD_EQ*(ideal_state: *i1*, ideal_state: *i2*, word32: *bd_ptr*) ≝
cppi_ram_word(*i1*, *bd_ptr*) = *cppi_ram_word*(*i2*, *bd_ptr*) ∧
cppi_ram_word(*i1*, *bd_ptr* + 4) = *cppi_ram_word*(*i2*, *bd_ptr* + 4) ∧
cppi_ram_word(*i1*, *bd_ptr* + 8) = *cppi_ram_word*(*i2*, *bd_ptr* + 8) ∧
cppi_ram_word(*i1*, *bd_ptr* + 12) = *cppi_ram_word*(*i2*, *bd_ptr* + 12).

- {word32}: *BD_ADDRESSES*(ideal_state: *i*, word32: *q*) returns the set of physical by addresses in CPPI_RAM that are allocated to all buffer descriptors in the queue pointed to by *q*.

  {word32}: *BD_ADDRESSES*(ideal_state: *i*, word32: *q*) ≝
  {*a* | ∃*bd_ptr* ∈ *queue*(*q*). *a* ∈ *bd_addresses*(*bd_ptr*)}.

- bool: *SAME_CPPI_RAM*(ideal_state: *i*, ideal_state: *s*, word32: *bd_ptr*) states that the the content of the buffer descriptors in the queue pointed to by *bd_ptr* is equal in *i* and *s*.

  bool: *SAME_CPPI_RAM*(ideal_state: *i*, ideal_state: *s*, word32: *bd_ptr*):
  ∀*a* ∈ *BD_ADDRESSES*(*bd_ptr*).
  i.nic.regs.CPPI_RAM(*a* – *CPPI_RAM*) =
  s.nic.regs.CPPI_RAM(*a* – *CPPI_RAM*)

- bool: *SAME_NIC_EXCEPT_CPPI_RAM*(ideal_state: *i*, ideal_state: *s*) states that *i* and *s* are equal except for the *CPPI_RAM* and the *oracle* state components.

  bool: *SAME_NIC_EXCEPT_CPPI_RAM*(ideal_state: *i*, ideal_state: *s*) ≝
  i.nic.dead_state = *s.nic.dead_state* ∧ *i.nic.interrupt* = *s.nic.interrupt* ∧
  i.nic.regs.DMACONTROL = *s.nic.regs.DMACONTROL* ∧
  i.nic.regs.CPDMA_SOFT_RESET =
  s.nic.regs.CPDMA_SOFT_RESET ∧
  i.nic.regs.RX_BUFFER_OFFSET =
  s.nic.regs.RX_BUFFER_OFFSET ∧
  i.nic.regs.TX0_HDP = *s.nic.regs.TX0_HDP* ∧
  i.nic.regs.RX0_HDP = *s.nic.regs.RX0_HDP* ∧
  i.nic.regs.TX_TEARDOWN = *s.nic.regs.TX_TEARDOWN* ∧
  i.nic.regs.RX_TEARDOWN = *s.nic.regs.RX_TEARDOWN* ∧
  i.nic.regs.TX0_CP = *s.nic.regs.TX0_CP* ∧
  i.nic.regs.RX0_CP = *s.nic.regs.RX0_CP* ∧
  i.nic.init_p = *s.nic. init_p* ∧
  i.nic.tx_p = *s.nic.*tx_p ∧ *i.nic.rx_p* = *s.nic.rx_p* ∧
  i.nic.tx_td_p = *s.nic.tx_td_p* ∧ *i.nic.rx_td_p* = *s.nic.rx_td_p*.

- bool: *UNUSED_TX_CPPI_RAM*(ideal_state: *i*, word32: *q*) states that CPPI_RAM bytes outside the queue *q* are unused by the transmission process of the NIC according to its configuration in state *i*, and that buffer descriptors are unmodified until associated memory reads have been performed. *q* must be a sub-queue of *i.oracle.tx0_active_queue*, in order for this statement to be true.

```
bool: $UNUSED\_TX\_CPPI\_RAM$(ideal_state: $i$, word32: $q$) $\overset{\text{def}}{=}$
    $\forall s \in$ ideal_state, $\pi \in NIC\_execution(s)$, $0 \le k < length(\pi)$,
            $pa, v \in$ word32.
        $SAME\_NIC\_EXCEPT\_CPPI\_RAM(i, s) \land$
        $SAME\_CPPI\_RAM(i, s, q) \land \pi_k \rightarrow_{nic\_memory\_read(pa, v)} \pi_{k+1}$
        $\Rightarrow$
        $\exists bd\_ptr \in$ word32.
            $bd\_ptr \in queue(q) \land$
            $pa[31:12] \in allocated\_tx\_blocks(i, bd\_ptr) \land$
            $[\forall 0 \le l \le k + 1.\ BD\_EQ(i, \pi_l, bd\_ptr)]$.
```

- bool: $UNUSED\_RX\_CPPI\_RAM$(ideal_state: i, word32: q) is similar as for transmission but with respect to reception.

```
bool: $UNUSED\_RX\_CPPI\_RAM$(ideal_state: $i$, word32: q) $\overset{\text{def}}{=}$
    $\forall s \in$ ideal_state, $\pi \in NIC\_execution(s)$, $0 \le k < length(\pi)$,
            $pa, v \in$ word32.
        $SAME\_NIC\_EXCEPT\_CPPI\_RAM(i, s) \land$
        $SAME\_CPPI\_RAM(i, s, q) \land \pi_k \rightarrow_{nic\_memory\_write(pa, v)} \pi_{k+1}$
        $\Rightarrow$
        $\exists bd\_ptr \in$ word32.
            $bd\_ptr \in queue(q) \land$
            $pa[31:12] \in allocated\_rx\_blocks(i, bd\_ptr) \land$
            $[\forall 0 \le l \le k + 1.\ BD\_EQ(i, \pi_l, bd\_ptr)]$.
```

The three main predicates that *NIC_BDS* rely on follow with a description of them. *CPPI_RAM_MODIFICATIONS_AND_MEMORY_ACCESSES* states that the queues pointed to by *tx0_active_queue* and *rx0_active queue* after they have been advanced to the nearest non-released SOP buffer descriptor, captures all memory accesses, and also all CPPI_RAM modifications.

```
bool $CPPI\_RAM\_MODIFICATIONS\_AND\_MEMORY\_ACCESSES$(ideal_state $i$)
    $\overset{\text{def}}{=}$
    $\forall s \in$ ideal_state, $\pi \in NIC\_execution(s)$, $0 \le k < length(\pi), pa, v \in$ word32.
        $UNUSED\_TX\_CPPI\_RAM(i, first\_non\_released\_sop(i,$
                                        $i.oracle.tx0\_active\_queue)) \land$
        $UNUSED\_RX\_CPPI\_RAM(i, first\_non\_released\_sop(i,$
                                        $i.oracle.rx0\_active\_queue)) \land$
        $\forall a \in [0x4A102000, 0x4A104000)$.
            $\pi_k.nic.regs.CPPI\_RAM(a - CPPI\_RAM) \neq$
            $\pi_{k+1}.nic.regs.CPPI\_RAM(a - CPPI\_RAM)$
            $\Rightarrow$
            $a \in BD\_ADDRESSES(i.oracle.tx0\_active\_queue) \cup$
                $BD\_ADDRESSES(i.oracle.rx0\_active\_queue)$.
```

This formula states that the transmission process of the NIC only uses the buffer descriptors beginning with the first non-released SOP buffer descriptor in the queue pointed to by *tx0_active_queue*. Similarly for the reception process. In addition, if a bit in CPPI_RAM is modified, then that bit belongs to a buffer descriptor in the queues pointed to by *tx0_active_queue* and *rx0_active_queue*. This include modifications made by the teardown processes.

Since the content of CPPI_RAM is unspecified outside the buffer descriptors beginning with the first non-released SOPs in *tx0_active_queue* and *rx0_active_queue*, it is known that the memory accesses the NIC can make in its current configuration are independent of this CPPI_RAM area, and therefore that *tx0_active_queue* and *rx0_active_queue* and *recv_bd_nr_blocks* capture all memory accesses the NIC can make in its current configuration. Since the memory accesses only depend on the buffer descriptors beginning at the first non-released SOP buffer descriptor of the queues, *tx0_active_queue* and *rx0_active_queue* can be updated to point to these buffer descriptors in the current state without missing any memory accesses.

To require that *tx0_active_queue* and *rx0_active_queue* point to buffer descriptors of the right type, *TX_PTR* and *RX_PTR* are used.

bool: $TX\_PTR(\text{ideal\_state: } i) \overset{\text{def}}{=}$
$\quad \forall \pi \in NIC\_execution(i), 0 \leq k < length(\pi).$
$\quad\quad [i.oracle.tx0\_active\_queue = 0 \Rightarrow$
$\quad\quad\; [\neg\exists pa, v \in \text{word32}.\ \pi_k \rightarrow_{nic\_memory\_read(pa,\, v)} \pi_{k+1}]] \wedge$
$\quad\quad [i\ .oracle.tx0\_active\_queue \neq 0 \Rightarrow$
$\quad\quad\; [\exists 0 \leq l \leq length(\pi).\ \forall l \leq m \leq length(\pi).$
$\quad\quad\quad cppi\_ram\_word(\pi_m, i.oracle.tx0\_active\_queue + 12)[31] = 1 \wedge$
$\quad\quad\quad cppi\_ram\_word(\pi_m, i.oracle.tx0\_active\_queue + 12)[29] = 0]].$

bool: $RX\_PTR(\text{ideal\_state: } i) \overset{\text{def}}{=}$
$\quad \forall \pi \in NIC\_execution(i), 0 \leq k < length(\pi).$
$\quad\quad [i\ .oracle.rx0\_active\_queue = 0 \Rightarrow$
$\quad\quad\; [\neg\exists pa, v \in \text{word32}.\ \pi_k \rightarrow_{nic\_memory\_write(pa,\, v)} \pi_{k+1}]] \wedge$
$\quad\quad [i\ .oracle.rx0\_active\_queue \neq 0 \Rightarrow$
$\quad\quad\; [\exists 0 \leq l \leq length(\pi).\ \forall l \leq m \leq length(\pi).$
$\quad\quad\quad cppi\_ram\_word(\pi_m, i.oracle.rx0\_active\_queue + 12)[31] = 1 \wedge$
$\quad\quad\quad cppi\_ram\_word(\pi_m, i.oracle.rx0\_active\_queue + 12)[29] = 0]].$

TX_PTR and *RX_PTR* states that for all states the NIC can produce by executing until it is idle from the current state *i*:

- If *tx0_active_queue* or *rx0_active_queue* is zero, then the NIC will not issue any memory reads or writes, respectively.
- If *tx0_active_queue* or *rx0_active_queue* is not zero, then a state will be reached from which the buffer descriptor pointed to by *tx0_active_queue* or *rx0_active_queue* is a released SOP, and will continue to be so unless the CPU interacts with the NIC.

Since:

- *CPPI_RAM_MODIFICATIONS_AND_MEMORY_ACCESSES* requires the buffer descriptors in the queues pointed to by *tx0_active_queue* and *rx0_active_queue* to capture all CPPI_RAM modifications and memory accesses, and
- the last modifications made by the NIC is to clear the ownership bit of the SOP buffer descriptor (the teardown processes of the NIC might not set the SOP bit, but according to *TD_STOP_NIC* does this mean that no memory

accesses will occur and therefore will the assumptions in *UNUSED_TX_CPPI_RAM* and *UNUSED_RX_CPPI_RAM* be false which means that these two predicates are still true),

the only way for *tx0_active_queue*/*rx0_active_queue* to point to a SOP buffer descriptor that follows a set of buffer descriptors that have not been completely processed by the NIC, is if the SOP and EOP buffer descriptors of that set already have all their bits set as the NIC would write them in their post-processing. This is not an issue, since the critical property is to capture all memory accesses.

To actually catch memory accesses, it must be proved that if memory is accessed during NIC execution, then it is because of a transition with the label *nic_memory_read* or *nic_memory_write*:

$$\pi_k \longrightarrow_{nic\_memory\_read(pa,\ v)} \pi_{k+1} \text{ or } \pi_k \longrightarrow_{nic\_memory\_write(pa,\ v)} \pi_{k+1}.$$

Then it is known, that all memory accesses are actually captured by *tx0_active_queue* and *rx0_active_queue*.

NIC_BDS is defined as follows:

bool: *NIC_BDS*(ideal_state: *i*) $\overset{\text{def}}{=}$
    CPPI_RAM_MODIFICATIONS_AND_MEMORY_ACCESSES(*i*) ∧
    TX_PTR(*i*) ∧ *RX_PTR*(*i*).

E.3.3 NO_BD_OVERLAPS

NO_BD_OVERLAPS:

- *No buffer descriptor in the queue pointed to by tx0_active_queue is overlapping any buffer descriptor in the queue pointed to by rx0_active_queue.*
- *No buffer descriptor in the queue pointed to by tx0_active_queue is overlapping any other buffer descriptor in the queue pointed to by tx0_active_queue, and similarly for rx0_active_queue.*

This property prevents the NIC from modifying *CPPI_RAM* in insecure ways.

```
bool: NO_BD_OVERLAPS(ideal_state: i) ≝
    ∀tx_q, rx_q ∈ {word32}.
        tx_q = queue(i, i.oracle.tx0_active_queue) ∧
        rx_q = queue(i, i.oracle.rx0_active_queue) ∧
        [∀tx_bd ∈ tx_q, rx_bd ∈ rx_q.
                    bd_addresses(tx_bd) ∩ bd_addresses(rx_bd) = ∅] ∧
        [∀bd1, bd2 ∈ tx_q. bd1 ≠ bd2 ⇒
                                bd_addresses(bd1]) ∩ bd_addresses(bd2) = ∅] ∧
        [∀bd1, bd2 ∈ rx_q. bd1 ≠ bd2 ⇒
                                bd_addresses(bd1]) ∩ bd_addresses(bd2) = ∅].
```

E.3.4 NIC_DATA_NO_EXEC_CONF

NIC_DATA_NO_EXEC_CONF:

All buffer descriptors in the queues identified by tx0_active_queue and rx0_active_queue only access memory blocks that are within Linux RAM. In addition, for rx0_active_queue:

- *Only non-executable data blocks are allowed to be accessible.*
- *The accessed memory addresses do not touch memory critical NIC registers. This property is implied by LINUX_PHYSICAL_MEMORY and IN_LINUX since data blocks are a part of Linux RAM and NIC registers are, of course, outside RAM.*

This property prevents the NIC from accessing non-Linux memory, writing unsigned code and reconfiguring itself.

Auxiliary functions:

- {word20}: *tx_block*(ideal_state: *i*) returns the set of block indexes accessed by the queue pointed to by *tx0_active_queue*.

  $$\{\text{word20}\}: tx\_block(\text{ideal\_state}: i) \overset{\text{def}}{=} \{bl \mid \exists bd\_ptr \in queue(i.oracle.tx0\_active\_queue).\; bl \in allocated\_tx\_blocks(i, bd\_ptr)\}.$$

- {word20}: *rx_block*(ideal_state: *i*) returns the set of block indexes accessed by the queue pointed to by *rx0_active_queue*.

  $$\{\text{word20}\}: rx\_block(\text{ideal\_state}: i) \overset{\text{def}}{=} \{bl \mid \exists bd\_ptr \in queue(i.oracle.rx0\_active\_queue).\; bl \in allocated\_rx\_blocks(i, bd\_ptr)\}.$$

$$\begin{aligned}&\text{bool}: NIC\_DATA\_NO\_EXEC\_CONF(\text{ideal\_state}: i) \overset{\text{def}}{=}\\ &\quad \forall bl \in \text{word20}.\\ &\qquad (bl \in tx\_block(i) \Rightarrow i.oracle.\tau(bl) \in \{L1, L2, D\}) \land\\ &\qquad (bl \in rx\_block(i) \Rightarrow i.oracle.\tau(bl) = D \land i.oracle.\rho_{ex}(bl) = 0).\end{aligned}$$

E.3.5 NIC_READ_ONLY

NIC_READ_ONLY:

Linux cannot execute NIC registers, nor write NIC registers that affect which memory accesses the NIC does.

This property prevents Linux from reconfiguring the NIC into an insecure state or executing unsigned code by interpreting the NIC registers (which the NIC can change as it wants) as instructions.

$$\begin{aligned}&\text{bool}: NIC\_READ\_ONLY(\text{ideal\_state}: i) \overset{\text{def}}{=}\\ &\quad \forall bl \in \text{word20}.\\ &\qquad [i.oracle.\tau(bl) = MN \Rightarrow i.oracle.\rho_{\text{wr}}(bl) = 0] \land\\ &\qquad [i.oracle.\tau(bl) \in \{MN, N\} \Rightarrow i.oracle.\rho_{ex}(bl) = 0].\end{aligned}$$

E.3.6 CANNOT_DIE

CANNOT_DIE:

The NIC will never run into a dead state in its current configuration.

This property ensures that the state of the NIC is defined, including after the NIC has made an arbitrary number of transitions. That is, the NIC is properly configured in its current state.

To make sure that the SOP, EOP and ownership bits are set as expected in the *NIC_BDS* predicate, and to also allow the oracle to update *tx0_active_queue* and *rx0_active_queue* correctly, the SOP, EOP and ownership bits must be correctly set in all buffer descriptors given to the NIC. This gives *CANNOT_DIE* another purpose: It implies, due to the non-dead state property that the SOP, EOP and ownership bits are correctly set in all buffer descriptors that can currently be operated on by the NIC. However, this property is only with respect to the initialization of buffer descriptors, and it does not state anything about what tx0_active_queue and rx0_active_queue points to, and therefore are *TX_PTR* and *RX_PTR* still necessary in *NIC_BDS*.

For reception, the setting of the SOP and EOP bits is done correctly by the NIC. For transmission, the setting of those bits must be done by software. Therefore, the software must correctly set the ownership bit for all buffer descriptors, and for transmission correctly set SOP and EOP bits such that they correctly delimit the buffer descriptors of a frame. This forces Linux to add all buffer descriptors of each frame to transmit simultaneously by linking together those buffer descriptors and then appending them to the transmission queue. Linux uses exactly one buffer descriptor for each frame so this requirement is not a practical problem.

CANNOT_DIE implies that all buffer descriptors used by the NIC have correctly set SOP, EOP and ownership bits. Consider the following steps of the NIC model:

- If the SOP and EOP bits does not match in the transmit queue, then the NIC enters a dead state (see *transmit_step3* and *transmit_step5*).
- If the ownership bit is not set in SOP transmit buffer descriptors, then the NIC enters a dead state (see *transmit_step3*).
- If the SOP and EOP bits are not cleared and the ownership bit is not set in a receive buffer descriptor, the NIC enters a dead state.

Since *CANNOT_DIE* states that the NIC never enters a dead state, none of these bits are incorrectly configured.

Another property that *CANNOT_DIE* implies is that the EOQ bit of a buffer descriptor in the queues pointed to by *tx0_active_queue* and *rx0_active_queue* is set if and only if that queue is emptied (see *transmit_step3*). Since EOQ bits are not allowed to be set by software, does this property enable a proof for that Lemma V holds for executions of *cppi_ram_handler*. That reasoning is described in Subsection C.4.2.

To summarize, this predicate guarantees that the following fields are correctly initialized:

- Transmission buffer descriptors:
    - SOP and EOP bits are matching.
    - Ownership bits are set in SOP buffer descriptors.

    - The buffer offset field is less than the buffer length field of SOP buffer descriptors.
    - The buffer length field is greater than zero, and the EOQ bit is cleared.
    - The accessed memory region corresponds to RAM.
    - The sum of the buffer length fields of a frame is equal to the packet length field of the SOP buffer descriptor.
- Reception buffer descriptors:
    - The buffer offset, SOP, EOP, are EOQ fields are zero.
    - The buffer length field is greater than zero, and the *RX_BUFFER_OFFSET* register.
    - The ownership bit is cleared.
    - Buffer descriptors not used by the currently processed received frame have their CRC flag set to zero.
    - The accessed memory region corresponds to RAM.

bool: *CANNOT_DIE*(*ideal_state*: *i*) $\overset{\text{def}}{=}$
$\forall \pi \in NIC\_execution(i), 0 \leq k \leq length(\pi). \neg \pi_k.nic.dead\_state.$

E.3.7 TD_STOP_NIC

TD_STOP_NIC:

If a buffer descriptor in the queues pointed to by tx0_active_queue and rx0_active_queue has the teardown bit set, then the corresponding NIC transmission or reception process is idle.

This property allows the oracle to securely clear *tx0_active_queue* and *rx0_active_queue* if a set teardown bit is encountered when updating *tx0_active_queue* and *rx0_active_queue*.

Auxiliary predicates:

- bool: *TD_SET*(ideal_state: *i*, word32: *bd_ptr*) returns true if and only if any buffer descriptor in the queue pointed to by *bd_ptr* has the teardown bit set in the state *i*.

```
bool: TD_SET(ideal_state: i, word32: bd_ptr):
    if $bd\_ptr = 0$ then
        return false
    else
        return $cppi\_ram\_word(i, bd\_ptr + 12)[27] = 1 \lor$
                    $TD\_SET(i, cppi\_ram\_word(i, bd\_ptr))$
```

TD_STOP_NIC is defined as follows:

bool: *TD_STOP_NIC*(ideal_state: *i*) $\overset{\text{def}}{=}$

$\forall \pi \in \mathit{NIC\_execution}(i), 0 \leq k < \mathit{length}(\pi), pa, val \in \text{word32}.$

$[\mathit{TD\_SET}(i, \mathit{tx0\_active\_queue}) \Rightarrow$

$[\neg\exists pa, v \in \text{word32}.\ \pi_k \rightarrow_{\mathit{nic\_memory\_read}(pa,\ v)} \pi_{k+1}]] \wedge$

$[\mathit{TD\_SET}(i, \mathit{rx0\_active\_queue}) \Rightarrow$

$[\neg\exists pa, v \in \text{word32}.\ \pi_k \rightarrow_{\mathit{nic\_memory\_write}(pa,\ v)} \pi_{k+1}]].$

E.3.8 RECV_BD_REF

RECV_BD_REF:

ρ_{NIC} is correct: $\rho_{NIC}(bl)$ is equal to the number of buffer descriptors in the queue pointed to by rx0_active_queue that can access the block bl.

bool: *RECV_BD_REF*(ideal_state: *i*) $\overset{\text{def}}{=}$

$\forall bl \in \text{word20}.$

$i.oracle.\rho_{NIC}(bl) = |\{(bl, \mathit{bd\_ptr}) \mid$

$\mathit{bd\_ptr} \in \mathit{queue}(i, i.oracle.\mathit{rx0\_active\_queue}) \wedge$

$bl \in \mathit{allocated\_rx\_blocks}(i, \mathit{bd\_ptr})\}|.$

For each physical block *bl*, it is paired with all buffer descriptors that are reachable from *rx0_active_queue* and accesses some byte in *bl* (the correctness of *recv_bd_nr_blocks* is established by *RECV_BD_NR_BLOCKS*). For each physical block *bl*, $\rho_{NIC}(bl)$ is equal to the number of those pairs. That is, the number of receive buffer descriptors in the queue pointed to by *rx0_active_queue* that specify a data buffer that addresses at least one byte in the block represented by *bl*.

E.3.9 INIT_TD_IDLE

INIT_TD_IDLE:

- *If initialized is true, then the NIC has been initialized properly and the NIC initialization process is idle.*
- *If tx0_hdp_initialized, rx0_hdp_initialized, tx0_cp_initialized or rx0_cp_initialized is true, then the NIC has completed its reset operation and the corresponding HDP or CP register has been zeroed.*
- *If tx0_teardingdown or rx0_tearingdown is false, then the corresponding NIC teardown process is idle.*

This property gives the oracle an accurate view of the status of the initialization and teardown processes of the NIC. It helps the oracle reject NIC register write requests that could otherwise make the NIC enter a dead state.

INIT_TD_IDLE is defined as:

bool: $\mathit{INIT\_TD\_IDLE}(\text{ideal\_state: } i) \overset{\text{def}}{=}$

$$
\begin{aligned}
&[\mathit{i.oracle.initialized} \Rightarrow \mathit{i.nic.init\_p.init\_complete} \land \mathit{i.nic.init\_p.init\_step} = 0] \land \\
&[\mathit{i.oracle.tx0\_hdp\_initialized} \Rightarrow \\
&\ \mathit{i.nic.regs.TX0\_HDP} = 0 \land \mathit{i.nic.init\_p.init\_step} \in \{1, 2\}] \land \\
&[\mathit{i.oracle.rx0\_hdp\_initialized} \Rightarrow \\
&\ \mathit{i.nic.regs.RX0\_HDP} = 0 \land \mathit{i.nic.init\_p.init\_step} \in \{1, 2\}] \land \\
&[\mathit{i.oracle.tx0\_cp\_initialized} \\
&\ \Rightarrow \mathit{i.nic.regs.TX0\_CP} = 0 \land \mathit{i.nic.init\_p.init\_step} \in \{1, 2\}] \land \\
&[\mathit{i.oracle.rx0\_cp\_initialized} \Rightarrow \\
&\ \mathit{i.nic.regs.RX0\_CP} = 0 \land \mathit{i.nic.init\_p.init\_step} \in \{1, 2\}] \land \\
&[\neg \mathit{i.oracle.tx0\_tearingdown} \Rightarrow \mathit{i.nic.tx\_td\_p.transmit\_teardown\_step} = 0] \land \\
&[\neg \mathit{i.oracle.rx0\_tearingdown} \Rightarrow \mathit{i.nic.rx\_td\_p.receive\_teardown\_step} = 0].
\end{aligned}
$$

This predicate makes it simple to analyze that the oracle handlers does not put the NIC in a dead state when writing certain registers, and it allows the oracle to correctly decide whether the NIC has been initialized or not. Actually, it must also be proved that these predicates prevent the NIC scheduler from scheduling these processes, in all NIC states that are produced by autonomous NIC transitions that follow the state *i*.

E.3.10 RX_BUFFER_OFFSET_DMACONTROL_ZERO

RX_BUFFER_OFFSET_DMACONTROL_ZERO:

The RX_BUFFER_OFFSET and DMACONTROL registers are zero.

This property prevents Linux from modifying *RX_BUFFER_OFFSET* to cause the NIC to enter a dead state during its processing of receive buffer descriptors, and it simplifies the NIC register write request handlers. It also prevents Linux from modifying the *DMACONTROL* register to prevent the NIC from entering a dead state.

The reason for having this predicate can be understood by considering the following scenario:

1. A buffer descriptor is added to the receive queue, which is accepted by the NIC register write request handlers since they check that the buffer length value of the added buffer descriptor is greater than the size of the *RX_BUFFER_OFFSET* register (otherwise the NIC enters a dead state).
2. Linux changes the *RX_BUFFER_OFFSET* register such that the new value is greater than or equal to the buffer length field of the previously added buffer descriptor.
3. The reception process of the NIC model detects that the buffer length field of the buffer descriptor added in step 1 is not greater than the *RX_BUFFER_OFFSET* register. This makes the NIC enter a dead state.

A solution to prevent this scenario of enabling the NIC to enter a dead state is to force the *RX_BUFFER_OFFSET* register to always be zero. This also allows *is_data_buffer_secure* (used as an auxiliary function to determine whether a buffer descriptor only accesses legal memory and does not cause a transition to a dead state) to only check if the buffer length field is greater than zero.

Since Linux does not use any other value than zero for the *RX_BUFFER_OFFSET* register, is this not a practical problem. The modeling of the *DMACONTROL* register is discussed in Section B.2.

bool: *RX_BUFFER_OFFSET_DMACONTROL_ZERO*(ideal_state: *i*) $\overset{\text{def}}{=}$
i.nic.regs.RX_BUFFER_OFFSET = 0 ∧ *i.nic.regs.DMACONTROL* = 0.

E.3.11 ACTIVE_CPPI_RAM

ACTIVE_CPPI_RAM:

α is correct: α(w) is true if and only if there is a buffer descriptor in the queues pointed to by tx0_active_queue and rx0_active_queue that occupies the wth 32-bit word of CPPI_RAM.

This property allows efficient checking of whether a *CPPI_RAM* write request is accessing a used part of *CPPI_RAM* or if a new queue is overlapping a queue that is already in use by the NIC. Therefore can *α* be used when deciding whether a CPPI_RAM write request shall be accepted or rejected.

bool: *ACTIVE_CPPI_RAM*(ideal_state: *i*) $\overset{\text{def}}{=}$
∀*bd_ptr* ∈ word32.
[*bd_ptr* ∈ *queue*(*i*, *i.oracle.tx0_active_queue*) ∪
queue(*i*, *i.oracle.rx0_active_queue*)
⇒
i.oracle.α((*bd_ptr* – *CPPI_RAM*) >> 2) = *true* ∧
i.oracle.α((*bd_ptr* – *CPPI_RAM* + 4) >> 2) = *true* ∧
i.oracle.α((*bd_ptr* – *CPPI_RAM* + 8) >> 2) = *true* ∧
i.oracle.α((*bd_ptr* – *CPPI_RAM* + 12) >> 2) = *true*]
∧
$\Sigma_{0 \leq j < 2048 \land i.oracle.\alpha(j) = true}\ 1 =$
|*queue*(*i*, *i.oracle.tx0_active_queue*) ∪ *queue*(*i*, *i.oracle.rx0_active_queue*)| · 4.

For all buffer descriptors in the queues pointed to by *tx0_active_queue* and *rx0_active_queue*, all of their four words are marked as *true* by *α*. Since the words are word aligned (by *FINITE_WORD_ALIGNED_CPPI_RAM_QUEUE*) they are logically right shifted two bits. Also, it is ensured that only these buffer descriptors are marked as *true* by making sure that the number of CPPI_RAM words *w* in *CPPI_RAM* that satisfies *α*(*w*) = *true* is equal to the number of buffer descriptors in the queues pointed to by the *tx0_active_queue* and *rx0_active_queue* multiplied by four (each buffer descriptor consists of four words). *BD_NO_OVERLAPS*, implies that no buffer descriptors overlaps, and therefore is this equation is correct.

E.4 State Component Dependences

The predicates in *S* depend on the following state components of the ideal state *i*:

- *FINITE_WORD_ALIGNED_CPPI_RAM_QUEUES*: *i.oracle.tx0_active_queue*, *i.oracle.rx0_active_queue* and *i.nic.regs.CPPI_RAM*.

- *NIC_BDS*: *i.oracle.tx0_active_queue*, *i.oracle.rx0_active_queue*, *i.oracle.recv_bd_nr_blocks* and *i.nic*.
- *NO_BD_OVERLAPS*: *i.oracle.tx0_active_queue*, *i.oracle.rx0_active_queue* and *i.nic.regs.CPPI_RAM*.
- *NIC_DATA_NO_EXEC_CONF*: *i.oracle.tx0_active_queue*, *i.oracle.rx0_active_queue*, *i.oracle.*τ, *i.oracle.*ρ_{ex}, *i.oracle.recv_bd_nr_blocks* and *i.nic.regs.CPPI_RAM*.
- *NIC_READ_ONLY*: *i.oracle.*ρ_{wt}, *i.oracle.*ρ_{ex} and *i.oracle.*τ.
- *CANNOT_DIE*: *i.nic*.
- *TD_STOP_NIC*: *i.nic*.
- *RECV_BD_REF*: *i.oracle.rx0_active_queue*, *i.oracle.*ρ_{NIC}, *i.oracle.recv_bd_nr_blocks* and *i.nic.regs.CPPI_RAM*.
- *INIT_TD_IDLE*: *i.oracle.initialized*, *i.oracle.tx0_hdp_initialized*, *i.oracle.rx0_hdp_initialized*, *i.oracle.tx0_cp_initialized*, *i.oracle.rx0_cp_initialized*, *i.oracle.tx0_tearingdown*, *i.oracle.rx0_tearingdown* and *i.nic*.
- *RX_BUFFER_OFFSET_DMACONTROL_ZERO*: *i.nic.regs.RX_BUFFER_OFFSET* and *i.nic.regs.DMACONTROL*.
- *ACTIVE_CPPI_RAM*: *i.oracle.tx0_active_queue*, *i.oracle.rx0_active_queue*, *i.oracle.*α and *i.nic.regs.CPPI_RAM*.
- *WT_EX_REF*: *i.oracle.*ρ_{wt}, *i.oracle.*ρ_{ex}, *i.oracle.*τ, and the content of *L1* and *L2* blocks.
- *SOUND_PT*: *i.oracle.*ρ_{wt}, *i.oracle.*ρ_{ex}, *i.oracle.*τ, *i.oracle.GI*, *i.oracle.sign*, and the content of *L1*, *L2*, and executable *D* blocks.
- *CONST_PT*: *i.oracle.*ρ_{wt}, *i.oracle.*τ.
- *SOUND_MMU*: *i.oracle.*τ, *i.cpu.cp15.TTBR0*, *i.cpu.cp15.DACR*, and the content of *L1*, *L2*, hypervisor code and monitor code blocks.
- *IN_LINUX*: *i.oracle.*τ, content of *L1* and *L2* blocks, *i.oracle.cpu.uregs.r15*, *i.cpu.cp15.TTBR0*, *i.cpu.cp15.DACR*[5:4] and *i.cpu.sregs.CPSR*[4:0].

Appendix F Sub-Level Lemmas

This appendix motivates the sub-level lemmas. Some of them are not used by the top-level lemmas but by other sub-level lemmas.

F.1 CPPI_RAM Write Lemma

The *CPPI_RAM Write Lemma* is:

If a trace consisting only of autonomous NIC transitions starts from an ideal state that satisfies S and writes a bit of a buffer descriptor in CPPI_RAM, then is the meaning of that write for that buffer descriptor in accordance with the NIC specification.

Reasoning: Assume that *S* holds. CPPI_RAM can either be changed by the NIC when it is post-processing buffer descriptors or executes the teardown processes, or by issuing memory write request to NIC registers. The latter form of CPPI_RAM writes cannot occur since the NIC enters a dead state before such memory write requests are issued. Therefore it is enough to reason about CPPI_RAM modifications due to post-processing of frames and teardown operations.

Because *NIC_BDS* and *NO_BD_OVERLAPS* hold, the buffer descriptors in the queues pointed to by *tx0_active_queue* and *rx0_active_queue* are the only ones that are potentially in use by the NIC and they do not overlap. This implies that when the NIC manipulates a buffer descriptor, it is only manipulating one buffer descriptor and that modification is because of the operation of the NIC. The NIC only manipulates buffer descriptors when post-processing transmitted or received frames or because a teardown process has made progress. That is, the modification is consistent with the NIC specification.

For instance, if a teardown bit is set in a buffer descriptor, then that bit is set because of a completed teardown operation, and not because some other bit was set in some other buffer descriptor because the two buffer descriptors overlapped.

F.2 Constant Memory Lemma

The *Constant Memory Lemma* is:

If an execution trace only consists of autonomous NIC transitions and emanates from an ideal state that satisfies S, then does no transition in that trace modify the content of an L1, L2 or executable D block, or hypervisor or monitor memory.

Reasoning: The *CPPI_RAM Write Lemma* and *NIC_BDS* imply that all memory writes are to addresses as specified by the buffer descriptors in the queue pointed to by *rx0_active_queue*. By *NIC_DATA_NO_EXEC_CONF* are those addresses only referring to non-executable *D* blocks, where ρ_{ex} is correct according to *WT_EX_REF*. According to *LINUX* are hypervisor and monitor memory blocks typed as $\perp$. Since a block can only have one type, does this imply that the NIC cannot change the content of *L1*, *L2* and executable *D* blocks, or hypervisor or monitor memory.

F.3 RM and IM Initially Related Lemma

The *RM and IM Initially Related Lemma* is:

Each initial state of the real model is related to some initial state of the ideal model:

$$\forall r \in RM.IS.\ \exists i \in IM.IS.\ r\ R\ i.$$

A description for how this lemma can be proved is as follows:

1. For a state $r \in RM.IS$ to be related to a state $i \in IM.IS$ the following must hold:

   $$r\ R\ i \wedge S(i) \wedge NIC\_INIT(i) \wedge ORACLE\_INIT(i) \wedge Linux\_exec(i).$$

   The last four conjuncts are from the definition of *IM.IS*. By expanding the definition of *R* and *S*, a new predicate *RM_INIT*(*r*) can be defined that is equivalent to the above formula, and which formalizes what *r* must satisfy in order to be related *i*:

   $RM\_INIT(\text{real\_state: } r) \overset{\text{def}}{=}$
   $\quad CPU\_EQ(r, i) \wedge MEMORY\_EQ(r, i) \wedge NIC\_EQ(r, i) \wedge$
   $\quad HM\_O\_EQ(r, i) \wedge$
   $\quad CPU\_MEMORY(i) \wedge NIC\_INIT(i) \wedge ORACLE\_INIT(i) \wedge$
   $Linux\_exec(i).$

   The predicate *NIC* of *S* is removed since the state components that *NIC* depends on are required to have specific values by *NIC_INIT* and *ORACLE_INIT*.

2. Let *START* be the predicate that states which state the CPU and the NIC are in, and what the content of memory is, when the real physical system is powered on. The NIC shall have the values as described by the comments in the definition of nic_state in Section B.3.

3. Let {real_state}: *boot_execution*(real_state: *r*) be the function that computes a set of states of type real_state. A state $r' \in boot\_execution(r)$ if and only if there exists a trace π that has been generated by the transition rules that defines the real model starting from a state $r = \pi_0$ and there exists $n \geq 0$ such that $\pi_n = r' \wedge [\forall 0 \leq k < n.\ \neg Linux\_exec(\pi_k)] \wedge Linux\_exec(\pi_n)$ holds.

4. Let *BOOT* be the predicate that states that the boot code execution of the hypervisor is correct:

   $$BOOT \overset{\text{def}}{=} \forall r, r' \in real\_state.$$
   $$START(r) \wedge r' \in boot\_execution(r) \Rightarrow RM\_INIT(r').$$

   The assumption of this predicate requires that the state:

   - *r* corresponds to a state where the physical computer has just been turned on, and
   - *r'* is the first state in a system execution where Linux is to execute its first instruction.

> This follows the intuitive description of the informal definition of *RM.IS*. The quantifier and the conclusion require that all initial states satisfy *RM_INIT*, and all initial states in *RM.IS* are related to an initial state of the ideal model.

If the predicate *BOOT* can be proved, then all initial states of the real model are related to some initial state of the ideal model, since *RM_INIT* implies that the state r' is related to some ideal state $i' \in IM.IS$. *BOOT* can be proved by computing the weakest precondition of *RM_INIT* in BAP with respect to the initialization code of the hypervisor and then proving in HOL4 that *START* implies that weakest precondition, provided that the initialization code does not interact with the NIC, as is done in [68]. However, the initialization code of the hypervisor initializes the NIC. Perhaps this can be solved by verifying the NIC interaction in HOL4 and the rest with BAP.

F.4 MMU Lemma

The *MMU Lemma* is:

If r R i holds for real and ideal model states r and i, then does the MMU operate identically in the states r and i:

$$
\begin{aligned}
&\forall r \in RM.S, i \in IM.S.\\
&\quad r\ R\ i\\
&\quad \Rightarrow\\
&\quad \forall pl \in \{PL0, PL1\}, va \in word32, ap \in \{rd, wt, ex\}.\\
&\qquad mmu(r, pl, va, ap) = mmu(i, pl, va, ap).
\end{aligned}
$$

Reasoning: The operation of the *mmu* depends on *TTBR0*, *DACR*, *CPSR*, and the page tables in memory. According to Lemma II, $S(i)$ holds:

- *SOUND_MMU*: *TTBR0* points to an *L1* block.
- *SOUND_PT*: Second-level links in *L1* blocks point to *L2* blocks.
- *IN_LINUX*: Blocks of type *L1* and *L2* are in Linux memory.

Since *R* requires that *TTBR0*, *DACR*, *CPSR* and Linux memory to be equal in *r* and *i*, the MMU operates identically in *r* and *i*.

F.5 Exceptions Preserve R Lemma

The *Exceptions Preserve R Lemma* is:

If r R i ∧ Linux_state(r) holds for real and ideal model states r and i, and r raises an exception in its next CPU transition, then i will also raise an identical exception in its next CPU transitions and the generated states are related by R:

$$
\begin{gathered}
\forall r, r' \in RM.S, i \in IM.S.\\
r\ R\ i \wedge Linux\_state(r) \wedge r \rightarrow_{EXC} r' \Rightarrow \exists i' \in IM.S.\ i \rightarrow_{EXC} i' \wedge r'\ R\ i'.
\end{gathered}
$$

Reasoning: Since $r\ R\ i \wedge Linux\_state(r)$ holds, does it mean that both *r* and *i* are in non-privileged mode in which the CPUs in *RM* and *IM* behave identically by their definitions. There are six different exceptions to consider. First are FIQ and IRQ interrupts considered. FIQ interrupts cannot occur and only the NIC can raise IRQ interrupts. Since *r R i* holds, are the *nic* state components identical in *r* and *i*, meaning that both have their interrupt flag raised. Since *R* requires the *CPSR* registers to be equal between *r* and *i*, does both CPUs have the same values of the *I* flag and hence both CPUs take the interrupt exception which modifies the states *r* and *i* identically.

The next two exceptions to consider are supervisor call and undefined instruction exceptions. These exceptions are raised due to specific instruction executions in non-privileged mode. By the *MMU Lemma* will the MMUs in *r* and *i* operate identically, and since *R* requires the program counters to be equal, the MMUs will compute the same results. Since a memory abort exception did not occur (since it is assumed that a supervisor call or undefined instruction exception occurred), did *mmu* return a value *pa* ∈ word32. This means that the MMU succeeded with its translation without any memory aborts. Therefore is it the instruction located at physical address *pa* that determines the outcome of the next transitions from *r* and *i*.

It is now reasoned which blocks *pa* can potentially belong to. *NIC_READ_ONLY* (cannot execute NIC registers according to ρ_{ex}), *WT_EX_REF* (correct ρ_{ex} with respect to *L1* and *L2* page tables), *SOUND_PT* (*L1* second-level entries point to *L2* blocks), *SOUND_MMU* (the MMU uses an L1 block as first-level page table), and *IN_LINUX* (unmapped or no access permissions to ⊥ blocks in *L1* and *L2* blocks), imply that *pa* does not belong to a block of type ⊥, *MN* or *N*. That is, *pa* belongs to an *L1*, *L2* or *D* block.

By *IN_LINUX* does these blocks belong to Linux memory, which in turn is equal in *r* and *i* by *R*. *R* also requires user mode registers and *CPSR* to be equal in *r* and *i*, which implies that the same instructions with the same operands will be tried to be executed in the next operations from the states *r* and *i*. This together with equal operations of the CPUs in non-privileged mode cause these instructions to raise the same exceptions which in turn will modify the state components identically.

Next to consider are prefetch abort exceptions. Since *R* requires equality of the program counters and *CPSR*, and by the *MMU Lemma*, instructions at the same locations are tried to be fetched with the same access permissions making the MMUs operate identically. Therefore does the same exceptions occur when *r* and *i* shall perform their next CPU transitions, which modify the states identically.

The final exceptions to consider are data aborts. Faults that are not related to access permissions (like MMU translation or memory system errors) are reasoned as in the case of prefetch abort exceptions. Therefore is focus on memory read and write access violations. As reasoned in the case of supervisor call and undefined instruction exceptions, will the next transitions from *r* and *i* try to execute the same CPU instructions with the same operands. This and the *MMU Lemma* means that these instructions will make the same memory references upon which the MMUs will operate identically in *r* and *i*. Therefore will the same exceptions be raised and *r* and *i* will be modified identically.

Irrespectively of which kind of exception that occurs in the transition from *r*, the next CPU transition from *i* will cause the same exception, and these exceptions will modify the state components of *r* and *i* identically, implying that *r' R i'* holds since *r R i* holds.

F.6 No NIC Interrupt Assumption

The *No NIC Interrupt Assumption* is not a lemma but an assumption about the exception handlers of the oracle:

The exception handlers of the oracle does not clear the I flag of CPSR until the CPU is returned to Linux.

F.7 HVM NIC Register Dependence Only Lemma

The *HVM NIC Register Dependence Only Lemma* is:

Consider a sub-trace $\pi[j{:}m]$ that corresponds to a transition $\pi_j \rightsquigarrow_{real} \pi_m$ of any exception handler execution of the hypervisor and the monitor, where π_j is related by R to some i ∈ IM.S, then in this trace, excluding NIC register accesses:

- *The transitions of the hypervisor and the monitor depend only on resources not modified by the NIC.*
- *The transitions of the NIC depend only on resources not modified by the hypervisor and the monitor.*
- *The resources that the hypervisor and the monitor modify are disjunct from the resources that the NIC modify.*

Reasoning: Apart from NIC registers, the hypervisor and monitor and the NIC can only affect each other through memory and interrupts. By the *No NIC Interrupt Assumption*, the NIC cannot affect the CPU through interrupts.

Consider how the NIC accesses memory. *First of all, the NIC does not depend on memory content* (see the NIC model function *memory_request_byte_reply* which does not read the received value of the memory read request reply). Let *r* be the start state of the handler trace and *i* the related state. Since *S*(*i*) ∧ *r R i* holds, the following is known about the state *r* due to the equality of the *cpu*, Linux memory, *nic*, and hypervisor plus monitor and *oracle* state components of *r* and *i*:

- The *CPPI_RAM Write Lemma*, which relies on *NIC_BDS* and *NO_BD_OVERLAPS*, implies that the NIC modifications to CPPI_RAM can never change the memory that the NIC is configured to access.
- *NIC_DATA_NO_EXEC_CONF* (NIC access only non-executable *D* blocks according to ρ_{ex} and τ), *WT_EX_REF* (ρ_{ex} is correct with respect to *L1* and *L2* blocks), *SOUND_PT* (second-level entries of *L1* blocks only point to *L2* blocks), and *SOUND_MMU* (*TTBR0* point to *L1* block) imply that that the NIC can only write non-executable *D* blocks, of which NIC registers do not belong to according to *IN_LINUX*. This prevents the NIC from reconfiguring itself or writing hypervisor or monitor memory.

The NIC register write request handlers are the only handlers that write NIC registers, and they *never* break any of the predicates in the bullets above. The reason is that the queues pointed to by *tx0_active_queue* and *rx0_active_queue* are written before the HDP registers are written or when those queues are extended by CPPI_RAM writes, and they never write state components that the *CPU_MEMORY* predicate depends on. This is an unmotivated additional lemma that is based on investigation of the oracle handlers *tx0_hdp_handler*, *rx0_hdp_handler* and *cppi_ram_handler*, which are the only handlers that configures which memory the NIC can access. *The conclusion is that the NIC is always configured to only write non-executable D blocks.*

Furthermore, *S*(*i*) ∧ *r R i* also imply:

- *SOUND_PT* and *SOUND_MMU* states that the MMU only uses *L1* and *L2* blocks, and that the memory of the hypervisor and the monitor is mapped to its allocated RAM area. Since *τ* only maps each block to one type, *L1* and *L2* blocks are distinct from NIC writable *D* blocks.
- *IN_LINUX* states that *L1* and *L2* blocks are a part of Linux RAM, and hence they cannot be represented by NIC registers.

These two bullets mean that NIC cannot change the virtual to physical address mapping of the hypervisor and the monitor.

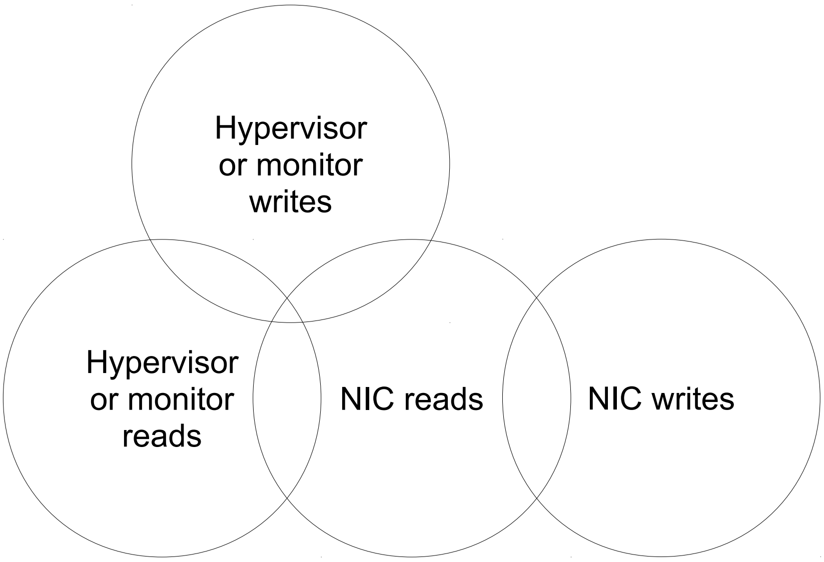


Figure 47: How the hypervisor, the monitor and the NIC access memory. Each circle represents one memory region. It is shown that the hypervisor and monitor only read data that cannot be modified by the NIC, and that the hypervisor and monitor and the NIC write disjunct parts of memory. Also, the operation of the NIC does not depend on memory content. These properties imply that the hypervisor and monitor and the NIC cannot affect each other through memory.

Consider now how the exception handlers access memory. Apart from their own memory do the memory mapping request handlers also access *L1*, *L2* and *D* blocks. Before *createL1*/*createL2* sets a block as *L1*/*L2* it must be non-writable by the NIC, and it is after that, that they and the other memory mapping request handlers read or write *L1*/*L2* blocks. Also, the memory mapping request handlers require the *D* blocks to be non-writable by the NIC in order to validate their signatures. *Hence, the exception handlers only depend on and write hypervisor, monitor, L1, L2, and non-NIC writable D blocks.*

Now it has been reasoned that:

- The transitions of the NIC do not depend on memory content, and it only writes non-executable *D* blocks.
- The hypervisor and the monitor does only depend on and write hypervisor, monitor, *L1, L2*, and non-NIC writable *D* blocks.

These two properties are illustrated in Figure 26 and imply:

- The NIC is independent of memory that the hypervisor and monitor can write.
- The hypervisor and monitor are independent of memory that the NIC can write.
- The hypervisor and monitor and the NIC write disjunct parts of memory.

Since memory is the only way for the hypervisor plus monitor and the NIC to affect each other, except for NIC registers (which are excluded by assumption) and interrupts (which are blocked), does this conclude the reasoning.

F.8 CPU and NIC Rescheduling Lemma

The *CPU and NIC Rescheduling Lemma* is:

Consider a sub-trace $\pi[j{:}m]$ that corresponds to a transition $\pi_j \rightsquigarrow_{real} \pi_m$ of an exception handler execution of the hypervisor and the monitor, where $\pi_j\ R\ i$ holds for some $i \in IM.S$. Assume $\pi[j{:}m]$ contains a consecutive sequence of two transitions. The first transitions is a CPU transition that is not the first or the last transition of the handler trace and which does not access a NIC register. The second transition is an autonomous NIC transition. That is $\pi_k \rightarrow_{CPU} \pi_{k+1} \rightarrow_{NIC} \pi_{k+2}$. The implication of this is that there exists a hypervisor and monitor handler sub-trace $\upsilon[j{:}m]$ such that the order of the operations of these two transitions are reversed, $\upsilon_k \rightarrow_{NIC} \upsilon_{k+1} \rightarrow_{CPU} \upsilon_{k+2}$, with all states in $\upsilon[j{:}m]$ being equal to the corresponding states in $\pi[j{:}m]$, except for π_{k+1} and υ_{k+1}:

$$\forall \pi \in RM.\Pi,\ i \in IM.S,\ a,\ e,\ a < b < e - 1.$$

$$\pi_{[a]} \rightsquigarrow_{real} \pi_{[e]} \land$$

$$CPUL(\pi_{[a]}) \land CPUL(\pi_{[e]}) \land EHT(\pi, a, e) \land$$

$$\pi_{[a]}\ R\ i \land$$

$$label(\pi, b) = CPU \land label(\pi, b{+}1) = NIC$$

$$\Rightarrow$$

$$\exists \pi' \in RM.\Pi.$$

$$\pi'_{[a]} \rightsquigarrow_{real} \pi'_{[e]} \land label(\pi, b) = NIC \land label(\pi, b{+}1) = CPU \land$$

$$\pi[a{:}b] = \pi'\,[a{:}b] \land \pi[b{+}2{:}e] = \pi'\,[b{+}2{:}e].$$

and vice versa where the order of the CPU and NIC transitions are reversed.

Reasoning: First of all, the first and the last transitions of the trace, corresponding to exception and exception return from and to Linux, are not included which implies $j < k < k + 2 < m$. A motivation will now follow for how υ can be constructed:

1. Set $\upsilon_0 = \pi_0$.
2. Apply the same transition rules that were used to generate all transitions in $\pi[0{:}k]$ in the same order in $\upsilon[0{:}k]$. This is possible because all of these states are equal since the first states are equal and the same transition rules can be used to generate the same states (including non-deterministic transition rules by choosing the same non-deterministic value as when the transition rule was applied in π).

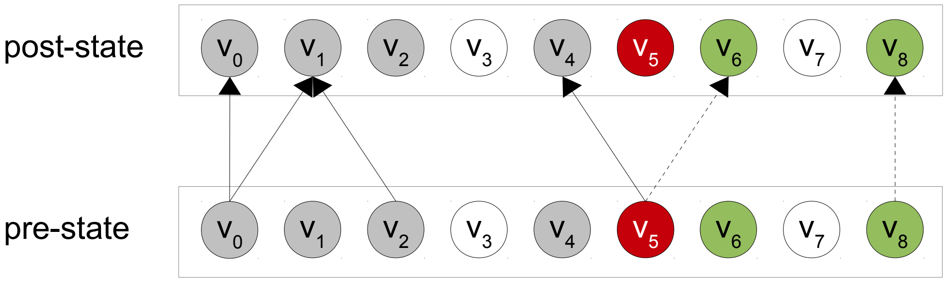


Figure 48: The independence of CPU and NIC transitions when the CPU does not access a NIC register. The real state is represented as nine state components (as an example). An arrow from a state component v_i to a state component v_j means that the operation of the next transition performed by the CPU or the NIC depends on the value of v_i when modifying the value of v_j. The state component dependencies of the next transitions of the CPU and the NIC are represented by continuous and dashed arrows, respectively. The next transition of the CPU depends on v_0, v_2 and v_5 and modifies v_0, v_1 and v_4, and the next transition of the NIC depends on v_5 and v_8 and modifies v_6 and v_8. Since the CPU and NIC transitions do not access common state components, except for v_5 which is constant, does it not matter in which order the CPU and NIC transitions occur.

3. By the *HVM NIC Register Dependence Only Lemma* and because the CPU transition does not access a NIC register, do the two CPU and NIC transitions operate independently. Figure 27 illustrates this reasoning. This means that:
   - The transition rule that was used to generate the NIC transition from the state π_{k+1} will also be applicable on the state π_k to generate the state υ_{k+1}, and it will modify the state identically as when producing π_{k+2}.
   - The transition rule that was used to generate the CPU transition from the state π_k will also be applicable on the state υ_{k+1} to generate the state υ_{k+2}, and it will modify the state identically as when producing π_{k+1}.

   Since the two transition rules do not write common state components, is υ_{k+2} equal to π_{k+2}.
4. Repeat step 2 from υ_{k+2} until υ_m is reached.

The same reasoning holds when the NIC transition precedes the CPU transition in π.

What remains to be shown is that $\neg Linux\_state(\upsilon_{k+1})$ holds in order to conclude that $\upsilon_j \rightsquigarrow_{real} \upsilon_m$ holds. There are two cases depending on the order of the CPU and NIC transitions:

- $\pi_k \rightarrow_{CPU} \pi_{k+1} \rightarrow_{NIC} \pi_{k+2}$ and $\upsilon_k \rightarrow_{NIC} \upsilon_{k+1} \rightarrow_{CPU} \upsilon_{k+2}$: Since NIC transitions do not modify *CPSR* or *DACR*, and since $\neg Linux\_state(\pi_k) \wedge \pi_k = \upsilon_k$ holds, $\neg Linux\_state(\upsilon_{k+1})$ also holds.
- $\pi_k \rightarrow_{NIC} \pi_{k+1} \rightarrow_{CPU} \pi_{k+2}$ and $\upsilon_k \rightarrow_{CPU} \upsilon_{k+1} \rightarrow_{NIC} \upsilon_{k+2}$: Since the CPU transition does not correspond to an exception return transition that returns control to Linux, which are the only transitions that produce states that satisfy *Linux_state*, and since $\neg Linux\_state(\pi_k) \wedge \pi_k = \upsilon_k$ holds, $Linux\_state(\upsilon_{k+1})$ holds.

F.9 NIC Preserves R Lemma

The *NIC Preserves R Lemma* is:

If a real model state r is related by R to an ideal model state i, and r makes an autonomous NIC transition to the state r', then i can make a corresponding transition to the state i' such that r' and i' are related by R:

$$\forall r, r' \in RM.S, i \in IM.S.\ r\ R\ i \wedge r \rightarrow_{NIC} r' \Rightarrow \exists i' \in IM.S.\ i \rightarrow_{NIC} i' \wedge r'\ R\ i'.$$

Reasoning: The definition of the autonomous NIC transition rules of the real and ideal models are identical and they only depend on and operate on the *nic* state components which are required to be equal by *R*. This, together with the assumption that *r* is related to *i*, imply that the corresponding transition rule of the ideal model:

- is also applicable to *i*, and
- can transform *i* to *i'* in an identical way as *r* was transformed into *r'*, by using the same non-deterministic value, if needed.

Since *r* and *i* are related by *R*, and the transition rules manipulate the related state components identically, *r'* and *i'* are also related by *R*.

F.10 Exception Handlers Implementations Assumption

The *Exception Handlers Implementations Assumption* states:

The exception handlers of the hypervisor and the monitor implement the design of the exception handlers of the oracle:

$$\begin{aligned}
&\forall r \in RM.S, i \in IM.S.\\
&\quad r\ R\ i \wedge mode(r) \neq usr\\
&\quad \Rightarrow\\
&\quad [\forall i' \in IM.S.\ i \rightarrow_{oracle} i' \Rightarrow \exists r' \in RM.S.\ r' \in handler\_execution(r) \wedge r'\ R\ i'] \wedge\\
&\quad [\forall r' \in RM.S.\ r' \in handler\_execution(r) \Rightarrow \exists i' \in IM.S.\ i \rightarrow_{oracle} i' \wedge r'\ R\ i'].
\end{aligned}$$

What follows is a reasoning of why this statement should be possible to prove, provided that the implementation follows the design: Let consider the first conjunct in the conclusion first. Since the oracle always takes control from privileged states and returns the CPU in non-privileged mode, the *cpu* state component in the state i is in the first state after an exception has occurred and i' is the result of the execution of an oracle exception handler.

Let *handler_execution* return the set of states that satisfy *Linux_state* in the sub-traces that occur when only CPU transitions are made from a given state where the *cpu* state component is in the first state after an exception (this includes NIC transitions due to NIC register accesses but no autonomous NIC transitions). Because $r\ R\ i$ holds, the *cpu* state component of r is also in the first state after an exception has occurred. This means that r is the state that the exception handlers of the hypervisor and monitor start their execution from, and r' where they end their execution. The only non-determinism in the oracle transition and the hypervisor and monitor exception handlers is because of NIC register writes to the HDP and CP registers. The *nic* state components are identical in r and i since they are related, and because of the assumption that the hypervisor and monitor operations follow the oracle design, do the hypervisor and the monitor perform the same operations as the oracle and all non-deterministic operations of the NIC that can be made in the oracle transition can also be made in the sub-trace of the exception handlers of the hypervisor and the monitor. Therefore are the end state r' and i' related.

The second conjunct in the conclusion is now considered. *handler_execution* is undefined for states in which the *cpu* state component is not in its first state after and exception. Therefore does the exception handlers of the hypervisor and the monitor start their execution from r. Since the *cpu* state component of r is in the first after an exception and r is related to i, is the *cpu* state component of i also in the first state after an exception. This means that the oracle can perform a transition from i into some state i'. The reasoning now is as for the other conjunct in the conclusion: The hypervisor and the monitor perform the same operations as the oracle, and the NIC can perform the same non-deterministic operations in both models. Therefore can the oracle modify the state i to produce the state i' in a corresponding way as the hypervisor and the monitor generates the state r' from r.

F.11 Exception Handlers Preserve R Lemma

If $\upsilon[k{:}m]$ *corresponds to a handler execution in the real model, where:*

- υ_k *is the start state, in which the CPU has not advanced since the last exception,*
- υ_m *is the end state, in which Linux is given the CPU after exception return (first state following* υ_k *that satisfies Linux_state),*
- *no autonomous NIC transitions occur in* $\upsilon[k{:}m]$*, and*
- $\upsilon_k\ R\ v_k$ *holds, where* v_k *is a state in the ideal model,*

then, there exists an oracle transition in the ideal model from v_k*,* $v_k \rightarrow_{oracle} v_{k+1}$*, such that R relates* υ_m *and* v_{k+1}*.*

Reasoning: Using the notation in the *Exception Handlers Implementations Assumption*, $\upsilon_k = r$ and $\upsilon_m = r'$, and hence $\upsilon_m \in$ *handler_execution*(υ_k). Since $\upsilon_k\ R\ v_k$ holds and the CPU is in non-privileged mode, can that assumption be used to conclude that $\exists v_{k+1} \in IM.S.\ v_k \rightarrow_{oracle} v_{k+1} \wedge \upsilon_m\ R\ v_{k+1}$ holds.

F.12 RLM Simulates ILM

The following three subsections motivates why transitions in $\rightsquigarrow_{ideal}$ should be possible to be matched by transitions in $\rightsquigarrow_{real}$, with a minor modification to the definition of transitions in $\rightsquigarrow_{real}$. That it, the roles of $\rightsquigarrow_{ideal}$ and $\rightsquigarrow_{real}$ in Lemma V are swapped.

F.12.1 Linux Transitions

Transitions in *RLM*, $r \rightsquigarrow_{real} r'$, that correspond to non-privileged CPU transitions, can be matched by corresponding transitions in *ILM*, $i \rightsquigarrow_{ideal} i'$, since the CPUs in *RM* and *IM* are deterministic, and non-privileged *ILM* transitions can match non-privileged *RLM* transitions. This gives:

$$\forall i, i' \in ILM.S, r \in RLM.S.\ i \rightsquigarrow_{ideal} i' \wedge r\ R\ i \Rightarrow \exists r' \in RLM.S.\ r \rightsquigarrow_{real} r' \wedge r'\ R\ i'.$$

F.12.2 Exception Handler Transitions

If the definition of $\rightsquigarrow_{real}$ is changed to allow autonomous NIC transitions after exception returns, (being equal to the definition of $\rightsquigarrow_{ideal}$), then privileged transitions in *RLM* and *ILM* can match each other:

- $\forall r, r' \in RLM.S, i \in ILM.S.$
  $r \rightsquigarrow_{real} r' \wedge r\ R\ i \Rightarrow \exists i' \in ILM.S.\ i \rightsquigarrow_{ideal} i' \wedge r'\ R\ i'.$
- $\forall i, i' \in ILM.S, r \in RLM.S.$
  $i \rightsquigarrow_{ideal} i' \wedge r\ R\ i \Rightarrow \exists r' \in RLM.S.\ r \rightsquigarrow_{real} r' \wedge r'\ R\ i'.$

The first property is proved as in Subsection 7.4.4.3 but where $\upsilon[j{:}m]$ has trailing autonomous NIC transitions after the exception return transition. These transitions need not be reordered, and when constructing $v[j{:}l]$, a fourth step is added that simply repeats the second step for the trailing autonomous NIC transitions.

The second property is proved as follows. The given transition $i \rightsquigarrow_{ideal} i$ that corresponds to a sub-trace $v[j{:}l]$ of the following shape:

$$v[j{:}l] = v_j \rightarrow_{EXC} v_{j+1} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_k \rightarrow_{SPEC} v_{k+1} \rightarrow_{NIC} \ldots \rightarrow_{NIC} v_l$$

can be matched by the real handler subtrace:

$$\pi[j{:}m] = \pi_j \rightarrow_{EXC} \pi_{j+1} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \pi_k \rightarrow_{CPU} \ldots \rightarrow_{CPU} \pi_h \rightarrow_{RET} \pi_{h+1} \rightarrow_{NIC} \ldots \rightarrow_{NIC} \pi_m,$$

where $\pi_k \rightarrow_{CPU} \ldots \rightarrow_{CPU} \pi_h \rightarrow_{RET} \pi_{h+1}$ matches $\mathrm{v}_\mathrm{k} \rightarrow_{SPEC} \mathrm{v}_{\mathrm{k}+1}$ without any intermingled NIC transitions. An intuitive reasoning follows. Since non-privileged execution of the real and ideal CPUs are identical and that execution is deterministic, both traces start with the same exception, which preserves *R* (i.e. the reverse of the *Exceptions Preserve R Lemma*; see non-privileged CPU transitions above). Then both traces are constructed by applying the same NIC transition rules in the same order, each of which preserves *R* (i.e. the reverse of the *NIC*

Transitions Preserve R Lemma; see for autonomous NIC transitions below). The oracle and the CPU handler transitions perform the same operations and also preserve *R* (i.e. there reverse of the *Exception Handlers Preserve R Lemma* which can be proved by using the second conjunct in the conclusion of the *Exception Handlers Implementations Assumption*). The second sequence of the autonomous NIC transitions are matched in the same way as the first sequence.

F.12.3 NIC Transitions

The motivation behind the *NIC Preserves R Lemma* also holds for the case when the ideal model makes a NIC transition and the real model shall make a corresponding transition to preserve *R*. Then, since $Linux\_state(i) \wedge r\ R\ i$ holds, and that NIC transitions do not modify *CPSR* or *DACR*, $Linux\_state(r) \wedge Linux\_state(r')$ holds. This follows the definition of $\rightsquigarrow_{real}$, which gives:

$$\forall i, i' \in ILM.S, r \in RLM.S.\ i \rightsquigarrow_{ideal} i' \wedge r\ R\ i \Rightarrow \exists r' \in RLM.S.\ r \rightsquigarrow_{real} r' \wedge r'\ R\ i'.$$