Verification of DMAC Device Driver Operations in HOL4

Robert D Platt

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Freek Verbeek
Wenjie Xiong

May 7, 2024
Blacksburg, Virginia

Keywords: Linux Device Drivers, State Machines, HOL4 Modeling, Device Driver Verification, DMA

Verification of DMAC Device Driver Operations in HOL4

Robert D Platt

(ABSTRACT)

Modern computer systems require efficient data transfers involving memory in order to get the best possible performance. However, even the most optimized CPUs take too long to access memory regions, which takes time away from doing the typical computations that a CPU is designed to do. To solve this, Direct Memory Access (DMA) is used, which allows peripherals and other hardware accelerators, such as stand-alone DMA Controllers (DMACs), to read and write memory without CPU intervention. However, DMA introduces security problems in which attackers are able to leak data and overwrite critical system components by bypassing typical operating system security mechanisms. This thesis presents a case study to model as well as verify DMA device driver code in HOL4, which is an interactive theorem prover (ITP) used for machine-checked verification. This thesis verifies parts of Intel's IXGBE X550 device driver, which is a complex, 10 Gbit Network Interface Card (NIC). This verification takes the first significant step towards proving that the DMA device driver configures the DMA device such that it preserves memory isolation, which ensures that only memory that is intended to be readable and writable will be accessed. This thesis also provides a formal method to verify that a loop terminates under all possible cases. This can be used to further verify the correctness of a DMA driver. These contributions allow for the overall increased security of memory when using DMA device drivers that are verified by this approach, leading to the hindrance of attacks on systems utilizing DMA.

Verification of DMAC Device Driver Operations in HOL4

Robert D Platt

(GENERAL AUDIENCE ABSTRACT)

Modern computer systems use Direct Memory Accesses (DMAs) in order to offload the CPU from doing memory transfers. However, this poses the problem that the CPU is not able to monitor every memory access made through DMA. This can lead to attackers utilizing vulnerabilities in the device drivers used to perform DMA operations. This thesis addresses this problem by modeling and verifying properties of a device driver that will prove that the driver configures DMA such that it is isolated. This thesis also models and verifies a loop to ensure that it terminates, further verifying the correctness of a function in a device driver. These contributions are significant because they allow for increased security of a computer system's memory, reducing the likelihood of attacks.

Dedication

This thesis is dedicated to my family and friends as well as my mentors in the ECE department at Virginia Tech.

Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, who has supported me and guided me through my time throughout my research and degree journey. He has helped me improve my skills greatly in the work I have done as well as providing me with great mentorship and advice in pursuing my graduate degree.

I would also like to thank Dr. Jonas Haglund, who has been a great mentor and teacher throughout my time working with the Systems Software Research Group at Virginia Tech. He has spent countless hours dedicated to helping me progress with my research, guiding me through decisions, and giving general advice for my career doing research at Virginia Tech.

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under Contract No. N66001-21-C-4028 and by the US Office of Naval Research (ONR) under grant N00014-21-1-2523.

Contents

List of Figures

List of Abbreviations

API Application Programming Interface

BD Buffer Descriptor

CPU Central Processing Unit

DMA Direct Memory Access

DMAC Direct Memory Access Controller

FSM Finite-State Machine

GPU Graphics Processing Unit

HOL4 Higher Order Logic

I/O Input Output

IOMMU Input-Output Memory Management Unit

ITP Interactive Theorem Prover

NIC Network Interface Card

OS Operating System

UART Universal Asynchronous Receiver Transmitter

USB Universal Serial Bus

Chapter 1

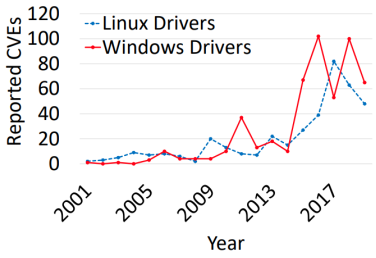
Introduction

The software development industry has been faced with the constant threat of malicious actors taking advantage of vulnerabilities that exist within software systems. Operating Systems (OSs), which are a major component of all computer systems, are a common target for these attackers. Attacks on OSs can be detrimental to the overall functionality of a system, and can result in data leaks, privilege escalation [3], kernel exploits [11] and complete system failures. This is obviously undesirable for most software systems, so a large effort has been continuously made by software engineers to circumvent these issues by increasing security as well as verifying that systems function correctly.

However, OSs are complex, and consist of many modules such as device drivers to interact with hardware devices. This presents the issue of device drivers containing vulnerabilities that would cause either the OS or the device itself to function maliciously, leaving an opportunity for attackers to exploit these opportunities. These device drivers are hardware-specific, thus, a separate effort must also be made to verify these device drivers in order to ensure that known vulnerabilities are as scarce as possible.

Device drivers make up approximately 70% of Linux [16], and often contain much more vulnerabilities than OSs, due to the fact that they are written by third-party device vendors. These vendors are not held to the same rigorous coding standards that OS developers such as Microsoft, Apple, and the Linux Community are, as they may not have the infrastructure to support meticulous testing and strict coding practices. Other factors such as time pressure,

stress, and lack of device documentation are also contributing factors to lower quality code. This results in device drivers containing many more vulnerabilities, often referred to as Common Vulnerabilities and Exploits (CVEs), than the rest of the OS. Figure 1.1 shows the total number of CVEs reported over time in device drivers in Linux and Windows, which shows that this is an increasing problem in OS security.



(a) CVEs, https://cve.mitre.org/.

Figure 1.1: Device Driver CVEs Reported Over Time [20]

Direct Memory Access (DMA) has also been introduced in devices in order to improve efficiency of the CPU. See Figure 1.2 for an example of the interactions between a device driver and a DMA Controller (DMAC).

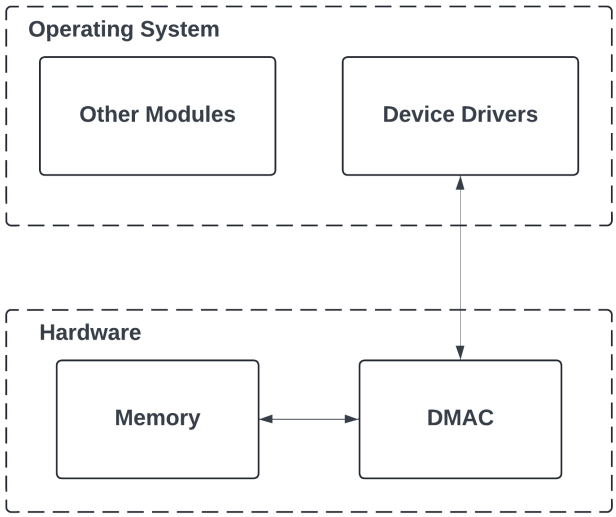


Figure 1.2: Device Driver Interaction with OS and Hardware

However, DMA adds to the complexity of ensuring security, since it can be used to leak data and to bypass OS security protocols such as user authentication. For example, a mobile GPU's DMAC has been used to obtain privilege escalation [8]. DMACs have also been used to crash Linux completely [19]. DMA malware has also been created, which can stealthily log keystrokes in Linux and Windows [26]. DMACs are configured using device drivers, which leaves a possibility for vulnerabilities within these drivers. This makes DMAC device drivers an attractive target for attackers, due to the fact that DMACs are able to access data without going through an OS. Verification of the device drivers as well as the DMAC is very important, as it can determine if a system could be compromised by an attacker in order for the developer to fix a vulnerability before it is discovered.

1.1 Motivation

DMACs provide many security problems, which attackers can possibly take advantage of and use for their own gain. This presents the important issue of DMAC device driver verification, which can be used to show that DMAC device drivers do not contain certain vulnerabilities by proving properties about them. The most important property pertaining to DMACs is memory isolation. Proving that a DMAC configuration preserves memory isolation would ensure that the DMAC is incapable of reading or writing memory that was not originally intended to be read or written. This would mean that attacks such as data leaks and overwriting critical kernel code are no longer possible using the DMAC. Unfortunately, there is a lack of significant work in verifying complex device drivers involving DMACs. This means that many device drivers go into production unverified, leaving possible vulnerabilities for attackers to take advantage of.

This thesis aims to bridge the current gap that exists in verified device drivers by presenting

a formalized approach to modeling and verifying properties of memory isolation of DMAC device drivers. This is done by converting device driver C code into a representative model of DMAC-related operations of the driver in HOL4, which is a proof assistant for higher-order logic. In this thesis, we model Intel's IXGBE X550 10Gbit Network Interface Card's (NIC) corresponding device driver.

Once the model is created, it is then analyzed in order to verify the first necessary property in ensuring memory isolation using the HOL4 theorem prover. This property states that all buffer descriptors (BDs), which are data structures stored in memory used to communicate between the DMAC and the driver, hold addresses to data buffers within readable and writable memory. Proving this property is the first step in proving total memory isolation, which would ensure that the security of DMAC device drivers is increased, and the number of possible attacks on a system's memory is limited. The contributions of this thesis aim to advance efforts which would allow for the use of DMAC device drivers in security critical systems.

1.2 Contributions

This thesis presents a formal way to model and verify DMAC device driver configuration of BDs. Verification of BD configuration can be used to prove memory isolation of a DMAC. This is significant, as it results in the increased security of systems utilizing DMA by decreasing the amount of possible attacks. The following list outlines the contributions of this thesis:

- *Formalized Device Driver Modeling*: This thesis presents an approach to formally model device drivers in HOL4. The methodology allows for a structured interaction between

the device driver, OS, and device, as well as interleaved execution of these system elements. This allows for the formal verification of each of these individual systems in order to prove properties such as memory isolation and functionality.

- *Verification of BD Initialization*: A key property in verifying memory isolation through the configuration of buffer descriptors in DMAC device drivers is proved. This property states that every BD that is maintained by the driver contains a valid address corresponding to a data buffer.

- *Verification of Loop Termination*: This thesis presents a formal method to verify that a BD initialization loop terminates in HOL4. This methodology is used to verify that a loop within a BD initialization function is correct, further validating the correctness of the driver.

1.3 Thesis Organization

This thesis is organized in the following chapters:

- Chapter 2 provides an overview of the background necessary to understand the terms and systems utilized in the verification of device drivers in HOL4.

- Chapter 3 discusses previous related work that has been done in modeling and verifying device drivers, OSs, and DMACs.

- Chapter 4 describes the driver being modeled in this thesis, a general approach to modeling device drivers, and the model implemented for this thesis.

- Chapter 5 gives a formalized case study approach to proving properties of memory isolation pertaining to a DMAC device driver.

- Chapter 6 provides a case study for formally modeling and verifying termination of loops in HOL4.
- Chapter 7 summarizes the contributions of this thesis, discusses limitations, and mentions future work in the field of DMAC device driver verification.

Chapter 2

Background

The necessary knowledge required for understanding the modeling and verification of DMAC device drivers using HOL4 includes device drivers (Section 2.1), DMACs (Section 2.2), transition systems (Section 2.3), and HOL4 (Section 2.4).

2.1 Device Drivers

Computer systems are typically made of a collection of third-party devices, such as monitors, keyboards, NICs, DMACs, CPUs, GPUs, and more (see Fig. 2.1 for an example with NICs and USBs).

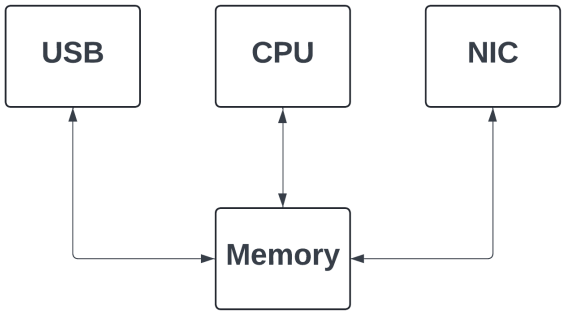


Figure 2.1: Computer Device Memory Organization Example

This means there is an extremely large number of possible combinations of different devices that can be integrated into one system. For this reason, it would be almost impossible to

write a singular operating system that is able to understand and control all of these devices on its own. To circumvent this problem and allow for more flexibility, OSs rely on device drivers, which are software modules written typically by device manufacturers that provide an OS with the ability to interact with varying hardware without needing to understand the specifics of these devices.

However, since the OS must rely on these complicated, third-party drivers and a lack of comprehensive hardware documentation, it is common for security flaws to exist within these drivers. For example, if a DMAC driver configures a DMAC incorrectly, it can cause the DMAC to access and edit memory that it does not have permission to edit, which could lead to a corrupt file or even operating system. This would be a critical security vulnerability in which a whole system could be compromised. For this reason, it is important that device drivers are analyzed thoroughly and rigorously, and contain as few bugs as possible.

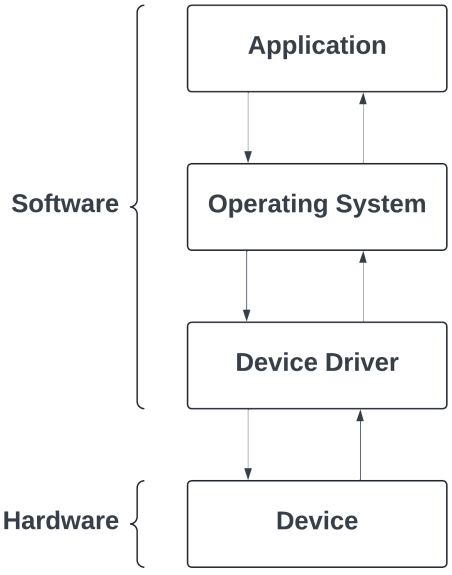


Figure 2.2: Device Driver and OS Interaction Diagram

Device drivers are a critical building block of modern computer systems. They facilitate the

interaction between hardware and OSs by leaving the software development to third-party vendors who know the hardware. However, this introduces many possibilities for security flaws, which can be very costly in many systems.

2.2 Direct Memory Access Controllers (DMACs)

Modern computer systems are optimized as much as possible to increase overall performance. One very common operation for computers and peripherals is interacting with memory. It also happens that memory accesses take much longer, often magnitudes longer, to perform than most other CPU instructions. This is due to the large amounts of data that are moved for each memory access. This time can be exacerbated by things such as cache misses, virtual address translations, moving from main memory to caches, etc. These memory access also take time away from CPUs that could be executing other program instructions. To address this problem, direct memory accesses were created, which allow for peripherals or other devices to access memory directly, instead of having to wait for the CPU to do it for them. An example of the time saved for a peripheral and CPU by doing this is shown in Fig. 2.3. This opens up the CPU to be able to do the more complicated computations that it was designed to do.

DMA Controllers (DMACs) are hardware devices that are integrated into computer systems to perform direct memory accesses for either peripheral devices or simply to perform memory-to-memory transfers in order to lessen CPU workloads. These devices take on the role of communicating with peripheral devices so that the CPU does not have to, thereby saving time for the CPU to do other work. DMACs must be initially configured by the CPU by writing to registers within the DMAC on startup. They also may need sections of memory to be initialized which will be used as buffer descriptors (BDs). These actions are done using

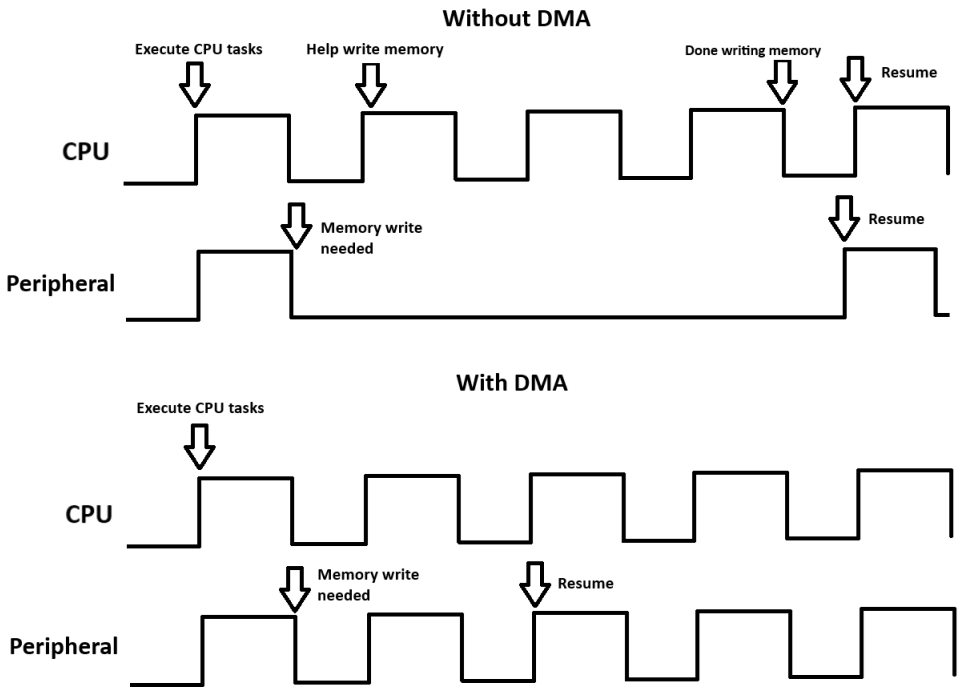


Figure 2.3: DMA Peripheral Cycles Saved

a DMAC device driver.

DMACs often use what are called BDs to communicate with the CPU and to read instructions for the memory operations that need to take place. These include receive a block, transmit a block, where to store data, and more. A simple example of a BD is shown in Fig. 2.4. If these BDs are configured incorrectly, the DMAC may access a section of memory that was not intended, leading to corrupt memory and possible leakage of data. These BDs are typically stored as a ring buffer (Fig. 2.5) in main memory, so that a DMAC can endlessly read instructions without running out of memory space.

CPU initialization of registers and BDs is very important, and it is the job of device drivers to do this. If the DMAC is initialized incorrectly, it can corrupt entire systems. However, if done correctly, they can also speed up systems immensely.

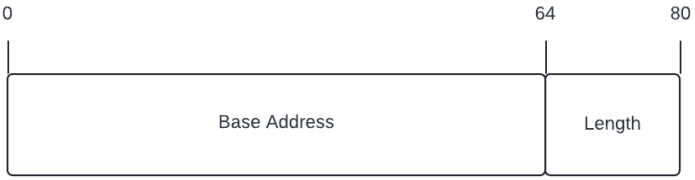


Figure 2.4: Buffer Descriptor Example

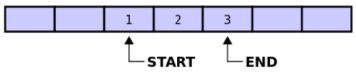


Figure 2.5: Ring Buffer Example With Three Elements [27]

2.3 Finite-State Machines

Finite-State machines (FSMs) are theoretical abstractions that can describe complex systems in terms of multiple states, which have clearly defined transitions between them. This allows for the analyzing of much smaller systems by looking solely at one state at a time. FSMs are used widely in software as well as hardware, due to their well-defined rules.

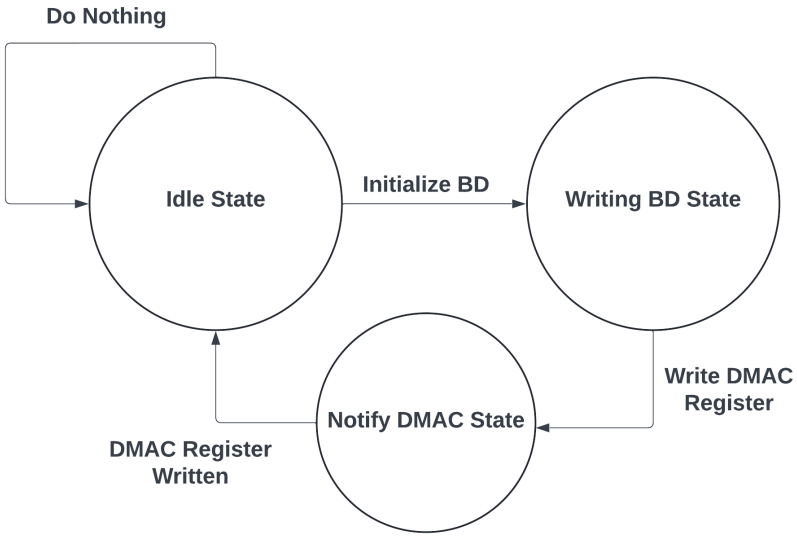


Figure 2.6: FSM Example with DMAC Driver

FSMs consist of every possible state that a system may be in, followed by every possible transition between these states. They are often displayed in graph form (as shown in Fig. 2.6), with the states being represented as nodes and the transitions represented as edges. This figure shows some example states in initializing a DMAC. The initial state is idle, then when a BD needs to be initialized, it moves to the "Writing BD State", then the "Notify DMAC State" when writing the DMAC register, and so on. This way, a visual representation of the system can be easily followed, allowing for a simpler understanding of systems.

In software, states can represent values of variables or locations in code. For example, states can be represented by a singular variable stored as an enumeration ("enum" in C) or a symbolic name, such as "#define IDLE 1" in C syntax. The transitions are then represented by cases in which this state variable is changed. For example, an if statement may be written so that when the transition conditions are true, then the state variable is changed to its next corresponding state. This allows for the simple translation of an abstract concept into code.

FSMs allow people to understand the intricate properties of large systems by breaking the overall system down into smaller transition systems. These systems are then translated into code or hardware, often with less errors due to the increased understanding of the system and formal methods.

2.4 Theorem Proving and HOL4

Formal verification and theorem proving have become popular in the software and hardware community. This is due to their ability to create models and describe properties of these models without any ambiguity. They are often used to check models for human errors, so that they can be proved beyond a doubt that a property holds true for certain actions and assumptions. However, proving this manually is often difficult and rigorous, which is why

interactive theorem provers (ITPs) are used to do a lot of the nuanced work of this process.

HOL4, the abbreviation for "Higher-Order Logic Theorem Prover", is a software tool in the specialized domain of ITPs. It is a language based on Standard Meta Language (SML), and was developed to allow for the proof and verification of complex systems and models using mathematics principles.

HOL4 uses strict axioms and rules to prove theorems about basic logic. For example, the "CONJ" rule combines two separate theorems by combining the assumptions from both theorems and making the conclusion of the resulting theorem the conjunction of the two conclusions. In HOL4, this would be notated as:

$$\frac{A \vdash C \qquad B \vdash D}{A, B \vdash C \wedge D}\ \text{CONJ}$$

In HOL4, theorems are expressed as a set of assumptions followed by "|-" and the conclusion. In this example, A, B, and A, B are all sets of assumptions, while C, D, and C /\ D are all conclusions (/\ is the symbol used for conjunction). The CONJ rule would be applied similar to a function that takes two inputs as theorems and outputs one theorem in the form shown above.

An axiom is a statement that defines an abstract structure or function. For example, one axiom used in HOL4 is:

$$\vdash \text{LET} = (\lambda\ \text{f x.}\ \text{f x})$$

This axiom defines the *LET* symbol, which allows for the substitution of values into variables that are used in expressions. This theorem uses lambda notation, which in this case denotes

a function that takes two parameters, f and x, where f is a function and x is some variable. These are then given as input to this lambda function and output as the function f applied to x. Another example of a lambda function's effect would be (\x y. x + y) 1 2 = (\y. 1 + y) 2 = 1 + 2 = 3.

If a theorem can not be constructed or broken into a combination of basic rules that are known to be true, then that theorem will not be proved. This means that any theorem that can be proved is factually true, allowing for people to make claims about what has been proved about their models.

To prove a theorem in HOL4, a "goal" must first be created. This goal is a statement that will eventually be proven using known theorems, rules, and assumptions. These goals can then be proven using "tactics", which are functions that will attempt to modify this goal using specific rules into a simplified goal. HOL4 has several default tactics, but they can also be created manually by defining a function that returns a new goal. Once a goal is simplified to a point where a tactic is able to prove it, the original goal has been proved.

2.4.1 Tactics in HOL4

Tactics are integral to proofs in HOL4 in that they provide a way to simplify the current goal into a simpler form. Tactics are functions that take a goal as input and return a set of subgoals as well as a validation function as a tuple. The goal is given in the form *(A, t)*, where A is the set of assumptions in the goal, and t is the conclusion to be proved from the given set of assumptions. *(A, t)* will then be simplified, assuming the tactic is successful and the validation function derives the given goal from the produced subgoal with a sequence of proof rules.

For example, if we would like to create a tactic that moves an antecedent A (representing a

statement that is either true or false), from an implication A ==> B (meaning if A is true, then B is true), to the assumptions (to the left of the turnstile symbol), an example of the use of this tactic may look like this:

$$\frac{\vdash A \wedge B \implies C}{A \wedge B \vdash C}\ \text{MY\_DISCH\_TAC}$$

This is often called discharging an implication, thus it is called MY_DISCH_TAC. As shown, $A \wedge B$ is moved to the assumptions of the goal, so that the conclusion of the goal is now simply C, a formula that is simpler than $A \wedge B \implies C$. Note that this tactic does exist in HOL's tactic library as DISCH_TAC (slightly differently), but for educational purposes an implementation will be shown here as well.

The implementation of MY_DISCH_TAC uses HOL functions such as dest_impl(), which splits the antecedent from the conclusion of an implication, to deconstruct and reconstruct a new assumption list and conclusion. The code below shows an example of the creation of these subgoals.

```
1  fun MY_DISCH_TAC (A, t) =
2      let val (ant, suc) = dest_imp t in
3      let val subgoals = [([ant], suc)] in
4      subgoals
5      end end
```

The *let* terms are used to define variables in functions, and work similar to the *LET* term explained in 2.4. The first let statement assigns a tuple of variables *(ant, suc)* to the destruction of the implication. Then, it creates a new list of subgoals, which is a list of pairs, where the first component is a list of assumptions (boolean formulas), and the second

component is the conclusion of the new goal (in this case there is only one pair). This example moves *ant* to the assumption list, and sets *suc* to the conclusion. The *end* terms simply close these *let* statements that were made earlier.

Now, all that is missing is the validation function that should be returned with this subgoal list. This function must take a list of theorems as input (which are the subgoals returned by MY_DISCH_TAC above) and returns a theorem as output (the theorem corresponding the the goal given as input to the tactic: A $\vdash$ t). The tactic must be verified by going backwards from the new theorems using known rules. In this simple example, the "DISCH" rule will be sufficient. This rule simply removes an assumption and creates an implication between this assumption and the conclusion (A $\vdash$ t is transformed to $\vdash$ A $\implies$ t). Therefore the validation function combined with the new subgoal function will look like this:

```
1            fun MY_DISCH_TAC (A, t) =
2                let val (ant, suc) = dest_imp t in
3                let val subgoals = [([ant], suc)] in
4                let val validation = fn thms =>
5                    let val thm = hd thms in
6                    DISCH ant thm
7                    end in
8                (subgoals, validation)
9                end end end
```

Some syntax to be aware of in this tactic is that validation is set equal to a function (*fn*) with *thms* as input, which is a list of theorems. Next, the *hd* function is called on this list of theorems in order to get the first or "head" element in this list.

It is important to note that this tactic does not implement error catching, which may occur when the conclusion of the goal is not an implication. This is ignored for the simplicity of this example, but should be implemented in an actual tactic library.

HOL4 is a powerful tool that allows users to prove qualities and properties about models and statements. It is often used to prove properties about hardware models as well as software systems. The critical component of HOL4 is its logical kernel that implements the proof rules. By minimizing the code size of this kernel, the correctness of HOL4 is maximized. These properties are undeniably true due to the fact that they are only proven using factually known lemmas and rules. On top of this, the proofs are machine-checked, avoiding the likelihood of human errors in manually checking large proofs. This means that users are able to rest assured that their systems are working as intended.

Chapter 3

Related Work

Both hardware and software has been considered for formal verification in past work. Work has been done in analyzing different computer systems with DMACs in order to verify that certain memory regions are isolated. However, many of these verification methods focus on these specific systems, but are not general enough to ensure correct DMAC configuration in all possible systems and require completely separate verification for other systems. Furthermore, they do not allow for the addition of a hardware model as well as an abstract OS model such that DMAC drivers can be verified without a completely new/significant analysis. This chapter discusses related work on the verification of memory isolation in DMAC drivers.

Section 3.1 mentions previous work done on the verification of OSs, Section 3.2 addresses previous attempts at verification of DMAC software, while Section 3.3 discusses modeling and verification of other types of device drivers.

3.1 OS Verification

Research has been done on the verification of OSs, especially smaller ones. For example, Klein *et al.* [18] were the first to fully verify functional completeness of an OS kernel. Klein *et al.* have designed a new microkernel named seL4, which is then formally verified using the Isabelle/HOL ITP to check for functional correctness. This microkernel was proven to

always follow an abstract specification, which includes properties in which it will never crash or perform unsafe operations. Although this approach may be feasible for a smaller OS code base such as a microkernel, it would not be feasible for a large OS with many device drivers such as Linux.

Nelson *et al.* [22] have also developed a fully verified kernel with some drivers called Hyperkernel. This is done with as much proof automation as possible using Microsoft's Z3 SMT automated theorem prover. Using this theorem prover, an OS kernel along with some drivers was designed, implemented and verified for functional correctness of many system calls. This was mainly intended for use when designing an OS kernel, instead of modeling and verifying an existing one.

Gu *et al.* [13] have extended previous efforts to create a fully-verified OS kernel by creating the first formally verified concurrent OS kernel. They have created an architecture for building certified concurrent OS kernels called CertiKOS, and have created a fully-verified concurrent OS kernel named mC2 using this tool. This is done using several abstractions along with interleaving execution, with the help of the Coq ITP. However, this tool can only be used to create new OS kernels, thus is not able to verify an already existing OS.

More recently, attempts have been made to extend these verified microkernels by verifying the systems that interact with them. An example of this is by Zhang *et al.* [28] who have fully verified the user-level API used to interact with the verified L4 micorkernel. They present 350 functional correctness and 39 safety properties, which are all verified using the Isabelle/HOL ITP. In the process of verifying this code, 10 bugs were found, all of which were provided fixes.

In addition to microkernels, some OS subsystems have also been verified. For example, Chen *et al.* [7] have successfully verified the FSCQ filesystem. Chen *et al.* have accomplished this

by creating a tool called Crash Hoare Logic (CHL), which checks that a filesystem recovers correctly according to its specification. The logic is verified using Coq.

3.2 DMAC Software Verification

There have been several successful attempts at proving isolated memory in software related to DMACs. One example of this is by Monniaux [21], who verified a device driver for a USB controller containing a DMAC. The automated tool used is able to detect memory access errors such as null pointer dereferences and arithmetic errors such as division by zero. However, this was done by creating C models of both the USB device as well as a model of the driver. The execution is described as an interleaved execution of two C threads (driver and dmac). Monniaux is able to verify that only certain memory locations are accessed by the model. The downsides of this approach is that a static analysis of a C model is unable to explicitly detect all possible execution scenarios and device interactions. It is also unable to create more abstract claims about the device driver's properties. An ITP, such as the one used in this thesis, would be able to verify in more detail and be able to consider every possible case to verify complete functional correctness and memory isolation.

HOL4 has also been used to model and analyze a system with DMA by Schwarz *et al.* [25]. In their paper, a previously created ARMv7 ISA model implemented in HOL4 is used to prove several isolation properties for devices which may use DMA. Several secure configurations of devices and isolation properties were identified. The method designed in this paper can be used to prove separation between two devices by using a hypervisor and separate guests. Although this does prove isolation properties and secure configurations, it does not specifically address DMAC device drivers and their configurations.

The closest work related to this thesis is that of Haglund *et al.* [14]. This work provides a

formal framework for how to model a DMAC in HOL4 in order to verify memory isolation. This framework is applicable to all DMACs, and allows the user to apply this model for any DMAC hardware. Although this work focuses mainly on hardware models and verification of memory isolation properties, it can be used to verify that a device driver correctly configures a DMAC in order to satisfy memory isolation. This is the main idea and motivation behind this thesis.

3.3 Device Driver Verification

There has also been work in the field of device driver modeling and verification. This section generalizes the last section, with all related work that does not discuss DMACs or memory isolation.

Alkassar *et al.* [2] have modeled and verified a UART device driver in the Isabelle/HOL ITP. In this work, a transition model of the CPU and the UART device were created so that they were able to interface with each other, and an assempler-level programming model was built in order to interact with the system via serial input. A simple driver for this model was then created and implemented in assembly language, which was then proven correct (for things such as termination). This focuses more on a formal model for the correct development of device drivers instead of verifying complex, existing drivers. Additionally, Alkassar *et al.* [1] created and verified a Isabelle/HOL model of a simple ATAPI hard drive device driver using Isabelle/HOL. This was the first formal functional verification of a device driver against a realistic device and system model.

Duan [10] has created an abstract device model, which was then instantiated by a serial port driver and paired with an existing ISA model implemented in HOL4. This model was verified as correct. Duan additionally expanded these models to support interrupts, and

created a formal model for verifying interrupt-based devices by basing states on CPU clock cycles, which are faster than peripheral cycles.

Chen *et al.* [5] have created a framework to formally verify interruptible OSs with device drivers using Coq. This is done by creating a concrete model as well as an abstract model and proving that they are equivalent.

There are also other examples of device driver verifications that did not use ITPs such as HOL4. For example, Kim *et al.* [17] have used model checkers to prove correctness of multi-sector read operations of a flash driver. Penninckx *et al.* [23] have used VeriFast, a software verifier based on separation logic to verify a Linux USB keyboard driver uses API correctly and does not contain illegal memory accesses or data races. Desai *et al.* [9] have verified that the USB hub driver in Windows 8 using a designed, domain-specific language named "P". This language is meant to help write asynchronous event driven code. Verification focuses mainly on the event handling of this driver. Tools that are used to check that device drivers comply with interfaces between OSs and drivers have also been created [4] [24].

Chapter 4

Device Driver Modeling

It is impossible to verify a device driver in an ITP such as HOL4 without first creating a model in HOL4 of how that driver works and interacts with other systems. This requires an in-depth study of the inner-workings of the driver and how it operates in relation to other components of a computer system such as the OS and the hardware. The device driver focused on in this thesis is complex, and manages a 10Gbit, high-performance network interface card (NIC), which is Intel's IXGBE-550 NIC driver (X550). Its device driver is over 10,000 lines of code, so a more abstract model is necessary.

This chapter discusses first a description of how the driver works in relation to DMACs in Section 4.1, then delves into how the specific driver is modeled in Section 4.2.

4.1 Driver Description

Due to the overall complexity and size of the X550 driver, it is not practical to model the entire system. It would also cause a large amount of effort devoted to proving non-critical aspects of the driver that are unrelated to memory isolation. For this reason, we focus on the parts of the driver related to DMAC operation, notably the initialization of buffer descriptors. This section will discuss how the driver interacts with the NIC as well as describe some of the hardware operation related to DMACs.

The most important parts of the driver when concerned with memory isolation are the buffer descriptors. This is how the driver and DMAC are able to communicate with each other, and determine where and how memory will be accessed by the peripheral. Correct or incorrect buffer descriptors will determine if the memory is isolated or not (assuming the OS and hardware operate correctly). For the X550, BDs are stored in the computer system's main memory. As a NIC, the X550 must deal with receiving as well as transmitting data quickly. Because of this, it splits descriptors into two main categories: *receive descriptors*, and *transmit descriptors*. We will focus on *receive descriptors*, since all that is applied to receive descriptors can be applied to transmit descriptors in a similar way.

Receive Descriptors

A *receive descriptor* is a BD used to hold information about where to store data when it is received by the NIC. This typically includes information about the allocated buffer addresses, checksums, packet types, statuses, and more. For example, when a packet is received from an outside source,the next descriptor in the receive queue 4.1 is read, and the received data is stored as indicated by the buffer address in the BD. The X550 driver supports two different types of receive descriptors: *legacy receive descriptors* and *advanced receive descriptors*.

Legacy Recieve Descriptors are simpler, 16-byte BDs that typically only store information such as the buffer start address and the length of the receive buffer. This is kept only for backwards-compatibility with older NICs. The X550 requires a much more optimized version of BDs in order to maintain speeds of up to 10 Gb/s, by implementing features such as receive side coalescing, segmentation offloading, etc. For this reason, we will look at *advanced receive descriptors* instead.

Advanced Receive Descriptors are more complex BDs data structures compared to legacy

descriptors. They contain information that allow for the X550 device to use more advanced networking features in order to speed up communication. These are also 16 bytes, however, they contain different fields depending on the state of the BD and whether the hardware or software is reading the BD. When the DMAC is reading the BD, it is interpreted in *Read format*. The layout of this BD is described in Fig. 4.1.

	63 1	0
0	Packet Buffer Address [63:1]	
8	Header Buffer Address [63:1]	DD

Figure 4.1: Field layout of advanced read format receive descriptors [15]

- *Packet Buffer Address*[63:0]: holds the physical address of the receive buffer in host memory.
- *Header Buffer Address*[63:1]: holds the physical address to a buffer within host memory containing the received packet's header, which includes checksums, Ethernet and TCP/IP headers.
- *DD*[0]: The descriptor done (DD) bit stands for "Descriptor Done" and is used by software to notify DMAC hardware that the descriptor is ready to be processed.

Write-Back Format is used when the driver is interpreting the BD after received data has been stored in the associated buffer and the BD is written by the DMAC. The fields of the write-back format of receive descriptors is presented in Fig. 4.2 and the relevant fields are explained below.

- *RSCCNT*[20:17]: indicates the number of coalesced packets that are stored in the buffer addressed by this descriptor.

	63 48	47 32	31	30 21	20 17	16 4	3 0
0	RSS Hash / Fragment Checksum / FCoE_PARAM / Flow Director Filters ID		SPH	HDR_LEN	RSCCNT	Packet Type	RSS Type
8	VLAN Tag	PKT_LEN	Extended Error		Extended Status / NEXTP		
	63 48	47 32	31 20		19 0		

Figure 4.2: Field layout of advanced write-back format receive descriptors [15]

- *HDR_LEN*[30:21]: indicates the size of the received packet's header written by the hardware.
- *Extended Status*[19:0]: Status information indicates whether the descriptor has been used and whether the referenced buffer is the last one for the packet.
- *Extended Error*[31:20]: States whether there is an Ethernet, IP or TCP header checksum error?
- *PKT_LEN*[47:32]: Packet length Specifies the number of bytes that constitute the received packet stored in the buffer addressed by this descriptor.

Receive Descriptor Queue

Receive descriptors are arranged as a circular buffer queue (ring buffer) in memory. Fig 4.3 shows the basic structure of this queue of receive descriptors.

The device driver upon initialization first allocates a chunk of memory big enough to hold all receive descriptors related to a single queue, then initializes the fields for a set of receive descriptors. Then, the *head* is set to the first available receive descriptor, and *tail* of this queue is set to the last available descriptor in the queue. The X550 keeps track of 128 separate receive queues, each of which are described by their set of registers outlined below.

- *Receive Descriptor Base Address Register*(RDBA): Indicates the starting physical ad-

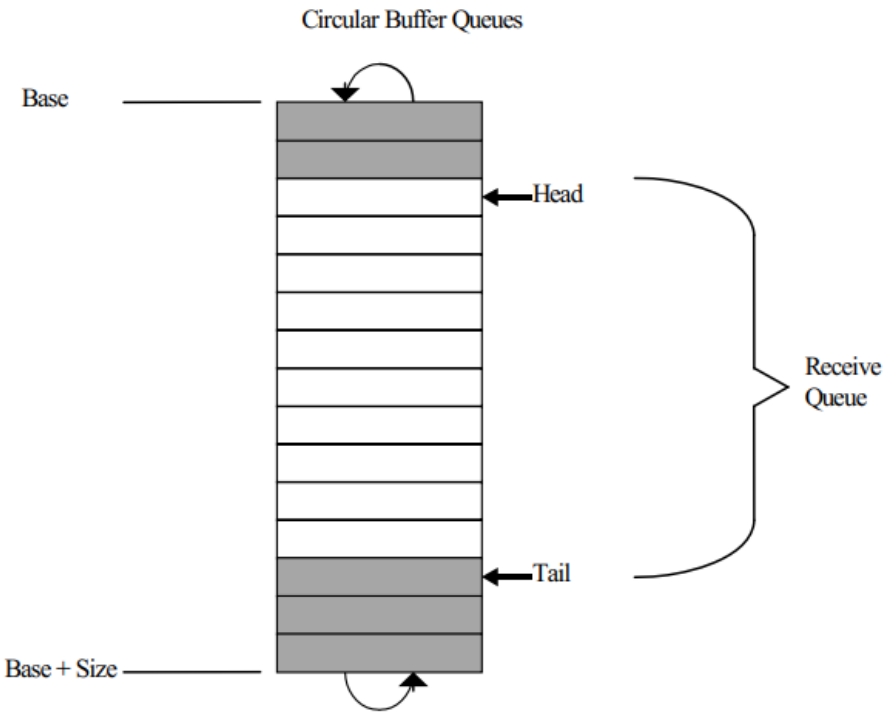


Figure 4.3: Receive Descriptor Queue structure [15]

dress of the descriptor ring buffer.

- *Receive Descriptor Length Register*(RDLEN): Indicates the total number of bytes allocated to the ring buffer.

- *Receive Descriptor Head Register*(RDH): Indicates the current in-progress descriptor (head).

- *Receive Descriptor Tail Register*(RDT): Indicates the physical address beyond the last descriptor hardware can use for reception.

Receive Data Flow

The packet receive process is described as the following steps:

1. The driver creates a descriptor ring by configuring x550's *RDBA*,*RDLEN*, *RDH* and *RDT* with the ring's address, length, head, and tail pointers, respectively
2. The driver initializes a descriptor by setting its buffer address field to a buffer referenced by an allocated memory address and placing it at the tail location.
3. The driver increments the queue's tail pointer to signal to the DMAC that additional descriptors have been added that the DMAC can use to store received packets.
4. A Packet is received from the network.
5. The DMAC fetches the head descriptor from the location indexed by the RDH register. If the packet data size exceeds the buffer data size, multiple descriptors are fetched by the DMAC.
6. The DMAC writes the received packet to the memory buffers addressed by the fetched descriptors.
7. The DMAC writes back the descriptor to signal to the driver the length of data stored and any possible errors.
8. The DMAC interrupts the device driver to notify it that a new packet is ready to be processed.
9. The driver reads and processes the buffer to forward the data to the TCP/IP stack for further processing.

4.2 Driver Modeling

This section describes the approach and implementation for modeling the X550 device driver as it relates to the DMAC. The overall model decisions are very important, as they determine

what can be proven about the actual driver. Section 4.2.1 will address the general DMAC device driver modeling approach. This is then followed by the implementation of the X550 driver model in Section 4.2.2.

4.2.1 Generic Modeling Approach

Manually modeling device drivers can be a difficult task, since it is important to capture every effect that an instruction has on the output of the driver, as well as what that instruction has done to the state of the driver. Drivers can also be large, for example, the X550 device driver is over 10,000 lines of code on its own. This makes it simply not feasible to model the device driver in its entirety. Since our focus is on memory isolation, we chose to model only the aspects of the driver that relate to configuring the DMAC and initializing BDs.

Since a device driver's main purpose is to facilitate communication between the device and the OS, the flow of data between these components must be modeled. This data flow is simulated by creating records in HOL4 (similar to C structs) that describe the input and output relationships between the kernel, driver, and device. Each of these records are explained below. Fig. 4.4 shows an example of the interaction between each of these abstractions using these records.

- *drv_input*: data received from the DMAC that is sent to the device driver.
- *drv_output*: data that will be written to the DMAC by the device driver.
- *kernel_input*: data that the operating system passes as arguments to the device driver.
- *kernel_output*: data that the device driver returns to the operating system.

Driver functions, often written in C, contain information such as local variables, control flow,

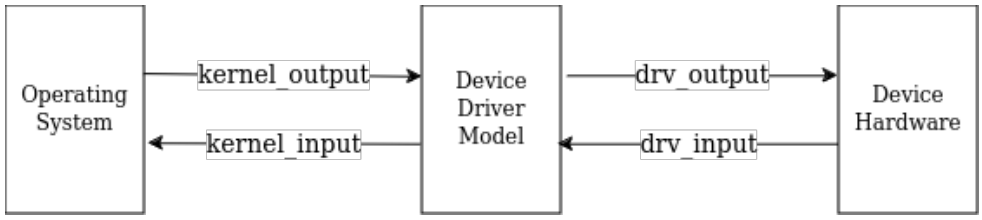


Figure 4.4: Record interface between driver model, OS, and DMAC hardware [12]

writes to hardware, memory reads, and more. These intricacies must be captured by the model in order to properly model the system. To capture the multiple interactions between hardware and the OS according to driver function implementations, we split up functions into several smaller functions. This way, the model can simulate interleaving execution between the OS, device, and driver. These smaller functions are called *control point functions*, as they occur between multiple control points within the original function. An example of a C function split into multiple control points is shown below.

```
1  void driver_func() {
2      // Control point 0
3      int data = register_read();
4      // Control point 1
5      int new_data = calculate(data);
6      global_var = new_data; // global variable
7      // Control point 2
8      register_write(new_data);
9  }
```

This C function is modeled as three separate executable steps, each of which can be interleaved with the DMAC hardware, to describe the parallel execution of the driver and the DMAC. Each step is described by one control point function, and the function as a whole is described by composing the control point functions in a scheduler function. This scheduler function will call its corresponding step depending on the current state of the driver. An

example of this is shown below.

```
1   void driver_func_cp0() {
2       int data = register_read();
3   }
4
5   void driver_func_cp1() {
6       int new_data = calculate(data);
7       global_var = new_data; // global variable
8   }
9
10  void driver_func_cp2() {
11      register_write(new_data);
12  }
13
14  void driver_func() {
15      if (driver_func__state == state__cp0)
16          driver_func_cp0();
17      else if (driver_func__state == state__cp1)
18          driver_func_cp1();
19      else if (driver_func__state == state__cp2)
20          driver_func_cp2();
21      else
22          return;
23  }
```

To allow data flow between the control point functions, the driver state is represented by a record, which contains all global variables of the driver, and all local variables and current control points of each modeled function. Fig. 4.5 shows a graphical representation of the finite state machine for this example.

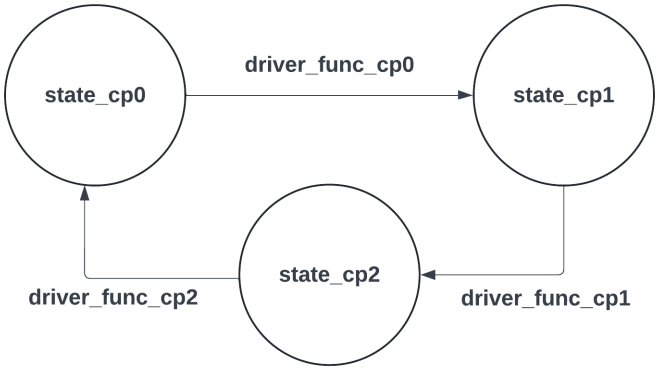


Figure 4.5: Finite State Machine for Control Point Example

Since one function (operation) is broken up into multiple functions (operations), this increases the number of possible hardware-software interleavings and gives a more accurate description of how the actual system operates. This means we are able to model every possible relevant state that exists in a driver's overall system.

Loops

Many functions written in the X550 driver use loops, however, HOL4 does not have a language construct that is similar to common programming language loops, such as in, for instance, C or Java. This raises the question of how to model loops. There are a couple of different ways to do this. One way is to use recursion, but this means that hardware operations cannot be interleaved with loop iterations as described in Section 4.2.1. Instead, we take advantage of the state machine format used for these functions. The start of a loop should be a new control point, and the first instruction after the loop will be another, separate control point. This can be seen in the example functions below, specifically with *driver_loop_func_cp1* and *driver_loop_func_cp2*. This means that after every iteration of the loop, it can check the condition required to break the loop and set the state accordingly.

An example of a function with a loop being modeled as a state machine with control points is shown below. The first code snippet shows the original function with the loop, while the second function shows the new, modeled version.

```
1  void driver_loop_func() {
2      int data = register_read();
3      int count = 0;
4      do {
5          count = count + 1;
6          data = data + 1;
7      } while (count < 10)
8
9      return data;
10 }
```

In a state machine model, this becomes:

```
1  void driver_loop_func_cp0() {
2      state.data = register_read();
3      state.count = 0;
4      state.state = state_cp1;
5  }
6
7  void driver_loop_func_cp1() {
8      state.count = state.count + 1;
9      data.count = data.count + 1;
10
11     if (state.count < 10)
12         state.state = state_cp1;
13     else
14         state.state = state_cp2;
15 }
```

```
16
17              void driver_loop_func_cp2() {
18                  return state.data;
19              }
20
21              void driver_func() {
22                  if (state.state == state_cp0)
23                      driver_loop_func_cp0();
24                  else if (state.state == state_cp1)
25                      driver_loop_func_cp1();
26                  else if (state.state == state_cp2)
27                      driver_loop_func_cp2();
28                  else
29                      return;
30              }
```

As the example shows, after each iteration of the loop, if the loop requires another iteration, it sets the control point state to the start of the loop (state_cp1). If the loop is done, go to the next control point state (state_cp2).

4.2.2 IXGBE-550 Driver Model

Now that the basics of our modeling approach has been explained, we will go into detail about our X550 device driver model. This model uses the basic approach as defined in Section 4.2.1, with a few additions that are specific to this driver. For example, this driver allocates and regularly writes the memory regions containing BDs. Since HOL4 does not support pointers to memory regions, a record representing a memory buffer called *r_mem_buffer_addr* is created. The datatype is outlined below in HOL4 code.

```
val _ = Datatype 'r_mem_buffer_addr =
<|
    address : 64 word;
    data : 32 word;
    data_region : 32 word list;
    data_len : 32 word;
    mapped : bool;
|>'
;
```

- *address*: the starting physical address for the memory buffer.
- *data*: Used when only one 32-bit word is stored in the buffer and contains that word.
- *data_region*:Only used if more than one 32-bit word is stored in the buffer, and is a list of 32-bit words stored in the buffer, with the word at list entry 0 being the first word located at the start address of the buffer, and the following entries containing the following words in the buffer.
- *data_len*: length of data in words (1 word = 4 bytes)
- *mapped*: indicates if the memory buffer has been mapped from a virtual address to physical for use by the DMAC.

These memory buffers are then referenced wherever pointers to memory are used in the actual driver. For example, the driver uses socket buffers, which are memory buffers with additional fields used to represent network packets. Note that some fields are left out, as they do not relate to DMAC operation and memory isolation. The implementation for a socket buffer struct is shown as *r_sk_buff_struct* below. This struct uses a slightly different

memory buffer model, in which and *address* field holds the start address, *buffer* holds the list of words within the buffer, and *buffer_len* holds the total number of words in the buffer.

```
val _ = Datatype `r_ixgbe_cb_struct =
<|
    skb_head : r_mem_buffer_addr;
    skb_tail : r_mem_buffer_addr;
    dma : r_mem_buffer_addr;
    append_cnt : 16 word;
    page_released : bool ;
|> `;

val _ = Datatype `r_sk_buff_struct =
<|
    address : 32 word;
    buffer : 32 word list;
    buffer_len : 32 word;
    cb : r_ixgbe_cb_struct;
|> ';
```

Next, these socket buffers are wrapped in structs with some additional fields (index, allocated) describing receive buffers to hold network packets. These are the most important buffers to look at when proving memory isolation, since they hold the physical address for the BDs used by the DMAC (*dma_addr*). This struct is represented by the *r_ixgbe_rx_buffer_struct* record below.

```
val _ = Datatype `r_ixgbe_rx_buffer_struct =
<|
    skb : r_sk_buff_struct;
    dma_addr : r_mem_buffer_addr;
    index : num;
```

```
6                   allocated : bool;
7               |> ';
```

Finally, as described in Section 4.1, these BDs are arranged as a ring buffer in main memory. The driver maintains these rings by storing their associated data structures (receive BDs) in the form of a linked list, as there are 128 separate rings. The datatype used to describe the C struct that stores ring-associated data is shown below as the *r_ixgbe_ring_element* record. The most relevant field that we will use to prove properties about memory isolation is the *rx_desc_list* field, as this is the list of all receive BDs to be used by the DMAC. An explanation of fields follows the record's definition.

```
1               val _ = Datatype `r_ixgbe_ring_element =
2               <|
3                   tx_buffer_info : r_ixgbe_tx_buffer_struct list;
4                   rx_buffer_info : r_ixgbe_rx_buffer_struct list;
5                   count : 16 word;
6                   size : 32 word;
7                   tx_desc_list : r_ixgbe_adv_tx_desc_struct list;
8                   rx_desc_list : r_ixgbe_adv_rx_desc_struct list;
9                   dma : r_mem_buffer_addr;
10                  next_to_use : 16 word;
11                  next_to_clean : 16 word;
12                  tail : 8 word;'
13              |>;
```

- *tx_buffer_info*: transmit buffer type information.
- *rx_buffer_info*: receive buffer type information.
- *count*: total number of BDs in this ring element.

- *size*: size (in bytes) of this struct/buffer.
- *tx_desc_list*: list of transmit descriptors in this buffer.
- *rx_desc_list*: list of receive descriptors in this buffer.
- *dma*: physical address of this buffer for use by the DMAC.
- *next_to_use*: index of the next buffer to use.
- *next_to_clean*: index of the next buffer to clean.
- *tail*: offset (in bytes) of tail from the base address.

Chapter 5

Buffer Descriptor Verification

Now that we have an abstract model of the DMAC related operations of the driver, we can analyze these operations to understand how the driver interacts with the hardware. We will look at two different properties of the driver to verify. First, in Section 5.2, we will prove that the buffer addresses given to all BDs in a ring buffer belong to a certain memory region, referred to in the following as ”valid”. Then, in Chapter 6, we will prove termination of a loop that initializes BDs in the driver.

In order to prove memory isolation, one important property that must be true is that BDs must hold addresses to buffers that exist in readable and writable memory [14]. This is a huge concern, because if a BD instructs a DMAC to access a memory location that the user is not given permission to access, it may be able to overwrite or read important assets such as the OS kernel or privileged information, or overwrite and corrupt critical code and data. This could lead to an attacker taking advantage of these vulnerabilities, leading to a successful attack on the system.

Section 5.2 will show that the initialization function used to allocate and initialize receive BDs in a ring buffer in the X550 driver will always produce valid read/write addresses, given some assumptions of the OS/Kernel. As mentioned previously, anything applied to receive BDs also applies to transmit BDs, so we will focus on Rx BDs in this thesis. This is the first step leading to a proof of memory isolation of a DMAC configured by a device driver.

5.1 Buffer Allocation Function

The main function used by the X550 driver to initialize Rx BDs is *ixgbe_alloc_rx_buffers*, which takes an *r_ixgbe_ring_element* (*rx_ring*) and a total count of buffers to initialize (*cleaned_count*). Given these arguments, it will loop though *cleaned_count* number of *rx_ring* BDs and will allocate memory for each BD's data buffer to store received packets in, (unless already allocated) then initialize each BD with the physical address to this buffer. It will also set the value of the *next_to_use* BD in the *r_ixgbe_ring_element* struct, which tells the driver which BD should be initialized next.

Using the modeling approach described in Section 4.2.1 as well as the records and states described in Section 4.2.2, we created a model of *ixgbe_alloc_rx_buffers*, which is separated into four control points: *state_ixgbe_alloc_rx_buffers_entry_cp0, state_ixgbe_alloc_rx_buffers_loop_cp1, state_ixgbe_alloc_rx_buffers_loop_cp2, and state_ixgbe_alloc_rx_buffers_end_cp3.* The control points are shown as comments on lines 3, 16, 27, and 43 in the code snippet below. The FSM describing these control points and their transitions can be seen in Figure 5.1.

```
1  void ixgbe_alloc_rx_buffers(struct ixgbe_ring *rx_ring, u16 cleaned_count)
2  {
3      /********************** CONTROL POINT 0 **********************/
4    union ixgbe_adv_rx_desc *rx_desc;
5    struct ixgbe_rx_buffer *bi;
6    //The index of the next descriptor that can be used by the NIC to store data
7    //in. Initialized to zero by sw_init.
8    u16 i = rx_ring->next_to_use; //0 immediately after initialization.
9
10   // nothing to do
11   if (!cleaned_count)
```

```
12      return;
13
14    rx_desc = IXGBE_RX_DESC(rx_ring, i); //Get descriptor i.
15    bi = &rx_ring->rx_buffer_info[i];  //Meta information about descriptor i.
16    //Decrement by the number of descriptors that the ring can store to get to
17    //first BD that has already been used to store data and that should be
18    //replaced/recycled. Initialized to 512.
19    //Immediately after initialization: 0 - 512 = -512 = 2^16 - 512.
20          i -= rx_ring->count;
21
22    do {
23          /********************** CONTROL POINT 1 **********************/
24          //If a page has already been allocated or is successfully allocated
25      //the current BD, then true is returned. Otherwise, false is returned
26      //and the loop is terminated.
27      if (!ixgbe_alloc_mapped_page(rx_ring, bi))
28        break;
29
30      //Little endian of physical address of where buffer data is stored?
31      rx_desc->read.pkt_addr = cpu_to_le64(bi->dma + bi->page_offset);
32
33          /********************** CONTROL POINT 2 **********************/
34      rx_desc++;  //Next descriptor.
35      bi++; //Next receive transfer meta information.
36      i++;  //Next BD index. Wraps around 2^16 = 64k desc.
37
38      // clear the status bits for the next_to_use descriptor
39      rx_desc->wb.upper.status_error = 0;
40
41      cleaned_count--;
42    } while (cleaned_count);
```

```
43
44      /********************** CONTROL POINT 3 **********************/
45    i += rx_ring->count;
46
47    if (rx_ring->next_to_use != i) {
48      rx_ring->next_to_use = i;
49
50      // update next to alloc since we have filled the ring
51      rx_ring->next_to_alloc = i;
52
53      //Sets the receive descriptor tail pointer to the current index.
54      //Immediately after initialization, i is 511.
55      writel(i, rx_ring->tail);
56    }
57  }
```

Control point 0 is the start of the function. Control point 1 is chosen as the start of the loop, so that it can be referred back to when the loop must be run again. Control point 2 was chosen because it is immediately after the data buffer is mapped and the BD is updated. This simplifies the proof process as well as leaves more possible execution interleaving. Control point 3 was chosen because it is immediately after the loop, so that it can be referred to when the loop must be broken.

5.2 Proof Strategy

We will refer to BDs that contain valid read/write addresses and lengths as "valid BDs". In order to prove that each BD in an *rx_ring* always contains a valid address, it must be shown that any possible state in the execution process contains a list of valid BDs. This requires

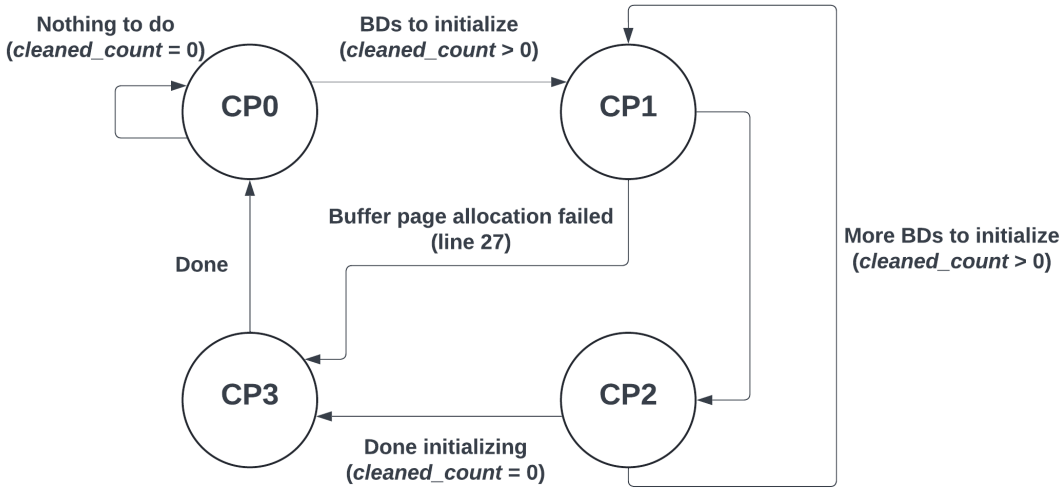


Figure 5.1: Finite State Machine for Control Points in *ixgbe_alloc_rx_buffers*

the assumption that all initial BDs are valid, since a single transition will not initialize all BDs in a ring. This is a justified assumption, since any BD that has not been initialized would not have been allocated, therefore not existing in the list. It is true that all BDs in an empty list are valid, therefore our assumption is valid. Note that it must be proven that the list is in fact empty upon startup, but this is left out from this discussion to focus on the more complicated part of BD initialization, rather than whether a list is empty or not.

With this assumption, we can prove that each execution step between two control points results in a driver state with a list of valid BDs. There are two possible cases: the list of BDs remains unchanged, or one of the BDs in the list is initialized. Proving the former case is trivial given the assumption described above. If all BDs are valid and they are unchanged, then all BDs are still valid. The latter case is more complex. This means that the updated BD must contain a valid address, which depends on the driver implementation, and that the other BDs are not changed. For this case, it also must be assumed that the OS kernel always returns valid addresses when allocating data buffers that are to be addressed by the BDs. Verification of the OS memory allocation module would require a complex model and

proof of the OS, which is outside the scope of this thesis.

Once it has been proven that each transition between two control points preserves that all BDs are valid, they can be combined to say that any possible invocation of the *ixgbe_alloc_rx_buffers* function will result in valid BDs. This is the first step towards proving memory isolation for a DMAC device driver, which is the goal of this thesis.

This section will go into depth on how the proof mentioned above is implemented in HOL4 with relation to the model that was created of the X550 driver. First, there are a few definitions that need to be made that will be used in these proofs. One of the most important ones is defining what a valid address is. In this case, actual valid addresses should not have an affect on the overall proof, since we will assume the kernel gives valid addresses when allocating memory. So, we choose some arbitrary values for concreteness for the range of valid addresses. The definitions in HOL4 are shown below, where *LOW_ADDRESS* is 16, *HIGH_ADDRESS* is 32, and *VALID_ADDRESS* states that all addresses of a buffer, starting at *base_address*, and having the byte size length are greater than or equal to *LOW_ADDRESS* and less than *HIGH_ADDRESS*.

```
1   Definition LOW_ADDRESS_DEF:
2     LOW_ADDRESS = 0x10w : 64 word
3   End
4
5   Definition HIGH_ADDRESS_DEF:
6     HIGH_ADDRESS = 0x20w : 64 word
7   End
8
9   Definition VALID_ADDRESS_DEF:
10    VALID_ADDRESS (base_address : 64 word) (length : 64 word) =
11    (LOW_ADDRESS <=+ base_address /\
12    base_address + length <+ HIGH_ADDRESS)
```

```
13          End
```

LOW_ADDRESS_DEF is the definition of the equality "*LOW_ADDRESS* = 0x10w : 64 word", which can then be used to substitute "0x10w : 64 word" in for any place where *LOW_ADDRESS* is used, similar to a macro in C.

Next, we can define our assumptions that will be used in the proofs. The function *ixgbe_alloc_mapped_page* maps and allocates the data buffers for storing received packets, see line 27 in 5.1. This function is modeled as a single transition (it should not interact with the DMAC, so this is a reasonable assumption, but complicated to prove since the DMAC can potentially affect the memory allocation software if the DMAC is not isolated from the memory allocation software). This function calls the Linux kernel functions *dev_alloc_pages* and *dma_map_page*, which belong to the memory management code in Linux and are assumed to be correct. These functions allocate a page for an rx data buffer and map it to a DMA physical address respectively. This results in the assumption in which any invocation of *ixgbe_alloc_mapped_page* gives a physical address that is valid. This assumption is as follows.

```
1     Definition DMA_VALID_ADDRESS_DEF:
2       DMA_VALID_ADDRESS  =
3       !buff buff' drv length.
4       buff' = ixgbe_alloc_mapped_page drv buff
5       ==>
6       VALID_ADDRESS (w2w (buff'.dma_addr.address + w2w buff'.page_offset))
7                     length
8     End
```

This definition states that *ixgbe_alloc_mapped_page*, when called with any driver state (*drv*) and initial buffer (*buff*), returns the modified buffer (*buff'*), whose dma address

(*dma_addr.address*) plus an offset (*buff'.page_offset*) is a valid address. In HOL4, the "!" symbol indicates a universal quantification, meaning that it considers all possible values of a variable, for instance, all 32-bit vectors of a variable of type :32 word. Words are HOL4's name for bit vectors. The "w2w" function in HOL4 simply converts a "word-to-word", meaning that smaller sized words would be cast to the largest word's size (in bits), with the additional most significant bits being zeros.

The next predicate states that the list of BDs (*bds*) addressing buffers of *length* bytes contain only valid BDs. This is shown below.

```
1  Definition EVERY_BD_VALID_DEF:
2    EVERY_BD_VALID bds length = EVERY
3                                (\bd. VALID_ADDRESS bd.r_pkt_addr.address
4                                 length)
5                                bds
6  End
```

Note that this definition states that every BD (*bd*) in *bds* addresses a buffer with only valid *bd.r_pkt_addr.address*, which is the receive packet's physical address stored in the BD. It also uses a given length, an argument that is set to a specific DMAC register. This definition uses lambda notation (the "\" symbol) explained in 2.4 as well as the *EVERY* term, which is defined as follows.

⊢ (!P. EVERY P [] = T) ∧ !P h t. EVERY P (h::t) = P h ∧ EVERY P t

This definition results in a theorem that simply defines *EVERY* as a function that takes a boolean function, *P* as a first argument, and a list as a second argument that is either empty ([]) or non-empty (h::t, where h is the first element and t is the following elements), and returns true if every element in that list (*h::t* or []) satisfies the predicate (*P*), and false otherwise.

With these definitions, we can now define our goals to be proven for each transition of the driver. An example of one of these goals used for transitions from the CP1 state can be seen below. This states that for all drivers, kernels, driver outputs, and length of BD lists; if the Linux kernel returns valid addresses, all initial BDs are valid, and the driver performs one step from the control point 1, ending up in control point 2 or 3 (see state diagram in Fig. 5.1), then the result is that the list of BDs is still valid.

```
1        !drv (k_out: r_kernel_output option) drv'
2        (drv_out : r_driver_output option) length.
3          DMA_VALID_ADDRESS /\
4          EVERY_BD_VALID drv.rx_ring.rx_desc_list length /\
5          (k_out, drv', drv_out) = ixgbe_alloc_rx_buffers_loop__cp1 drv
6          ==>
7          EVERY_BD_VALID drv'.rx_ring.rx_desc_list length
```

In the case where the driver execution is at CP0, the BDs are unchanged. This means we simply need to prove that *drv'.rx_ring.rx_desc_list* is equal to *drv.rx_ring.rx_desc_list*, which becomes true after executing the transition function.

In the case where the driver is at CP1, this is slightly more difficult, since one BD is changed in some cases. In the cases where a BD is not changed (kernel has failed to map a DMA address), we can simply use a similar proof to the one for CP0. In the case where a BD is updated, we need to show that this new BD is valid. To do this, we need to show that the new BD is initialized such that it satisfies the DMA_VALID_ADDRESS assumption. Since we assume that the function used to initialize these BDs always provides a valid address from the kernel, this will be true. It also must be proven that adding a valid BD to a list of valid BDs results in a full list of valid BDs. This is done using a theorem in the HOL4 listTheory library called IMP_EVERY_LUPDATE, which states that if an updated element in a list satisfies some predicate, and all other elements satisfy this predicate, then all elements in

the list will satisfy the predicate.

Cases CP2 and CP3 use the same proof strategy as CP0, since neither of these transition functions modify any BDs in the ring. *drv'.rx_ring.rx_desc_list* is proven to be equal to *drv.rx_ring.rx_desc_list* in both of these cases, proving the initial goal to be true.

Once all of these separate transitions have been verified, they can be combined to prove that the *ixgbe_alloc_rx_buffers* function does in fact always result in a ring of valid BDs. The final goal that is proved is shown below.

```
1        !drv length clean_cnt (k_out: r_kernel_output option) drv'
2        (drv_out : r_driver_output option).
3          DMA_VALID_ADDRESS /\
4          EVERY_BD_VALID drv.rx_ring.rx_desc_list length /\
5          (k_out, drv', drv_out) = ixgbe_alloc_rx_buffers drv clean_cnt
6          ==>
7          EVERY_BD_VALID drv'.rx_ring.rx_desc_list length
```

Chapter 6

Termination Verification

One important property to ensure that a driver operates correctly is that the functions and loops within them terminate. Although this does not prove memory isolation, it does prove that the driver does not get stuck in any infinite loops. This means that there are no cases in which the driver will not complete a function that it is required to complete, leading to a more secure driver that works as intended (and does not crash). This chapter will discuss how the function that initializes BDs is proven to terminate using HOL4.

6.1 Loop Model

The loop that will be verified is the one mentioned in Section 5.1, starting at control point 1. The loop section is also given below for convenience in Listing 6.1. This loop is used to iterate through the amount of BDs to be initialized for reception. It is a do...while loop, so it will execute at least once, then check if *cleaned_count* is zero. If it is, then the loop is done and the function moves on to the next control point. If not, the driver decrements *cleaned_count* by one, and goes from CP2 back to CP1 (start of the loop). This is also shown in Figure 5.1.

```
1     /********************** CONTROL POINT 1 **********************/
2   do {
3     if (!ixgbe_alloc_mapped_page(rx_ring, bi))
```

```
4         break;
5
6       /*
7        * Refresh the desc even if buffer_addrs didn't change
8        * because each write-back erases this info.
9        */
10      rx_desc->read.pkt_addr = cpu_to_le64(bi->dma + bi->page_offset);
11
12          /********************** CONTROL POINT 2 **********************/
13      rx_desc++;
14      bi++;
15      i++;
16      if (unlikely(!i)) {
17        rx_desc = IXGBE_RX_DESC(rx_ring, 0);
18        bi = rx_ring->rx_buffer_info;
19        i -= rx_ring->count;
20      }
21
22      /* clear the status bits for the next_to_use descriptor */
23      rx_desc->wb.upper.status_error = 0;
24
25      cleaned_count--;
26    } while (cleaned_count);
27
28      /********************** CONTROL POINT 3 **********************/
```

In order to prove termination of this function, the two control point functions from CP1 and CP2 within the loop will need to be combined. The specific reason for this will be explained later in Section 6.2. To do this, another function was created which checks the control point of the execution, which means that this function models execution between control point 1

and 3 or 2 and 3. This implementation is shown below.

```
val ixgbe_alloc_rx_buffers_loop_def = Define‘

  ixgbe_alloc_rx_buffers_loop (drv : r_driver_state) =
  if drv.state_ixgbe_alloc_rx_buffers__state.state =
  state_ixgbe_alloc_rx_buffers_loop__cp1 then
    let (k_out, drv', drv_out) =
    ixgbe_alloc_rx_buffers_loop__cp1 drv in
    if drv'.state_ixgbe_alloc_rx_buffers__state.state =
    state_ixgbe_alloc_rx_buffers_loop__cp2 then
      ixgbe_alloc_rx_buffers_loop__cp2 drv'
    else
      (k_out, drv', drv_out)
  else
    ixgbe_alloc_rx_buffers_loop__cp2 drv'
;
```

This allows for the execution of the loop section of this function in one call, regardless of its state. Now that this function is created, we can make a wrapper function that calls all control points sequentially just as the real driver executes *ixgbe_alloc_rx_buffers*. Proving that *ixgbe_alloc_rx_buffers*, terminates corresponds to the C loop being verified to terminate. The HOL4 implementation of this function is given below, where *cleaned_count* is the given number of BDs to initialize.

```
val ixgbe_alloc_rx_buffers_def = Define
‘
  ixgbe_alloc_rx_buffers (drv : r_driver_state)
  (cleaned_count : 16 word) =
  if drv.state_ixgbe_alloc_rx_buffers__state.state =
  state_ixgbe_alloc_rx_buffers_entry__cp0 then
```

```
7             ixgbe_alloc_rx_buffers_entry__cp0 drv cleaned_count
8           else if drv.state_ixgbe_alloc_rx_buffers__state.state =
9           state_ixgbe_alloc_rx_buffers_loop__cp1 \/
10                  drv.state_ixgbe_alloc_rx_buffers__state.state =
11                  state_ixgbe_alloc_rx_buffers_loop__cp2 then
12            ixgbe_alloc_rx_buffers_loop drv
13          else if drv.state_ixgbe_alloc_rx_buffers__state.state =
14          state_ixgbe_alloc_rx_buffers_end__cp3 then
15            ixgbe_alloc_rx_buffers_end__cp3 drv
16          else
17            (NONE : r_kernel_output option, drv,
18            NONE : r_driver_output option)
19        ‘;
```

6.2 HOL4 Implementation

Typically, when defining a function in HOL4, termination is automatically verified. However, some functions are too complex to prove termination automatically, therefore the user must give guidance and prove this manually. This is the case for the loop in *ixgbe_alloc_rx_buffers*, because the loop spans multiple states and the count variable (*cleaned_count*) that *ixgbe_alloc_rx_buffers* recurses on to determine how many BDs and buffers remain to initialize and allocate, which the loop relies on to terminate, is a variable that HOL4 does not keep track of. This means that termination of ixgbe_alloc_rx_buffers must be proven manually.

To prove termination, we need to be able to prove that each invocation of the body of a recursive function results in a decremented value. This would mean that the function does in fact terminate, since natural numbers (the type of the variable being decremented) cannot

be negative (the value that the recursion occurs on is getting smaller and smaller, but cannot be infinitely small). To do this, we must create a function that returns a natural number (non-negative integer) based on the state of the driver. We will call this the "measurement" function.

```
Definition measure_ixgbe_alloc_rx_buffers_def:
measure_ixgbe_alloc_rx_buffers driver_state =
  if driver_state.state_ixgbe_alloc_rx_buffers__state.state =
  state_ixgbe_alloc_rx_buffers_entry__cp0 then
    w2n driver_state.state_ixgbe_alloc_rx_buffers__state.cleaned_count + 2
  else if driver_state.state_ixgbe_alloc_rx_buffers__state.state =
  state_ixgbe_alloc_rx_buffers_loop__cp1 \/
  driver_state.state_ixgbe_alloc_rx_buffers__state.state =
  state_ixgbe_alloc_rx_buffers_loop__cp2 then
    w2n driver_state.state_ixgbe_alloc_rx_buffers__state.cleaned_count + 1
  else
    w2n driver_state.state_ixgbe_alloc_rx_buffers__state.cleaned_count
End
```

The measurement function has a value associated with each control point to each state. CP0 has a value of 2 added to the natural number corresponding to the bit-vector *cleaned_count*, CP1/CP2 (since they are executed together as discussed above) both have a value of 1 added to *cleaned_count*, and CP3 has a value of 0 added to *cleaned_count*. This means that when a transition from CP0 to CP1 or CP1 to CP3 occurs, the natural number is decremented by 1 (due to the state change), and similarly when a transition occurs from CP1 to CP2 and back to CP1 (due to the decrement of *cleaned_count* as defined by *ixgbe_rx_alloc_buffers*).

This function is then used to create a definition *MEASURE_IXGBE_ALLOC_RX_BUFFERS_DEF*, which compares the pre-state *driver_state1* before the transition to the post-state *driver_state2* after the transition, and returns true if the measurement function (*mea-*

sure_ixgbe_alloc_rx_buffers) of the post-state is less than the pre-state:

```
Definition MEASURE_IXGBE_ALLOC_RX_BUFFERS_DEF:
MEASURE_IXGBE_ALLOC_RX_BUFFERS driver_state2 driver_state1 =
  (measure_ixgbe_alloc_rx_buffers driver_state2 <
  measure_ixgbe_alloc_rx_buffers driver_state1)
End
```

The goal used to prove termination of *ixgbe_execute_alloc* is:

```
?R. WF R /\
drv k_out drv' drv_out.
  drv.state_ixgbe_alloc_rx_buffers__state.cleaned_count <> 0w /\
  (k_out,drv',drv_out) =
  ixgbe_alloc_rx_buffers drv
    drv.state_ixgbe_alloc_rx_buffers__state.cleaned_count /\
  drv'.state_ixgbe_alloc_rx_buffers__state.state <>
  state_ixgbe_alloc_rx_buffers_entry__cp0
  ==>
  R drv' drv
```

R in this case will be the measurement function we defined earlier as *MEASURE_IXGBE_ALLOC_RX_BUFFERS*. Since *measure_ixgbe_alloc_rx_buffers* only transitions between two control points (i.e. one transition) per call, we can prove this by proving that each control point's function satisfies this definition. This splits the goal into four separate cases (one for each possible control point: CP0, CP1, CP2, or CP3):

In the case where drv'.state_ixgbe_alloc_rx_buffers__state.state = state_ixgbe_alloc_rx_buffers_entry__cp0, the next state is either CP0 (if *cleaned_count is zero*) or CP1. If the next state is CP0, then the function has terminated, so this proves the goal to be true. If the next state is CP1, the measurement value is decremented by one due to the state change

(see *measure_ixgbe_alloc_rx_buffers_def*), making the conclusion and goal true.

In the case where drv'.state_ixgbe_alloc_rx_buffers___state.state = state_ixgbe_alloc_-rx_buffers_loop___cp1 or drv'.state_ixgbe_alloc_rx_buffers___state.state = state_ixgbe-_alloc_rx_buffers_loop___cp2, it must be proven that either the post-state is CP3 or *cleaned-_count* has been decremented. Looking at Figure 5.1, this is true, since the only possible next states (after combining CP1 and CP2) results in at least one of these. Therefore, both of these cases' goals are true.

Finally, in the case where drv'.state_ixgbe_alloc_rx_buffers___state.state = state_ixgbe-_alloc_rx_buffers_end___cp3, the next state will always be CP0. This can easily be proven, since one of the other assumptions is that drv'.state_ixgbe_alloc_rx_buffers___state.state is not state_ixgbe_alloc_rx_buffers_entry___cp0. This is because a state of CP0 would mean the function has terminated. Since these assumptions contradict each other, the set of assumptions will never be true, causing the goal to be true.

Chapter 7

Conclusions

This chapter summarizes the overall contributions of this thesis in Section 7.1, discusses the limitations of the work in Section 7.2, and mentions possible future work in Section 7.3.

7.1 Summary of Contributions

The following list outlines the contributions of this thesis:

- *Formalized Device Driver Modeling*: This thesis presents a systematic approach to modeling DMAC drivers in HOL4. The methodology allows for a formal interaction between the device driver, OS, and device, as well as interleaved execution of each element of the system. This allows for the formal verification of each of these individual systems in order to prove properties such as memory isolation and functional correctness.

- *Verification of BD Initialization*: A key property in verifying memory isolation through the configuration of buffer descriptors in a Linux NIC DMAC driver is proved. This property states that every BD that is maintained by the driver contains a valid address corresponding to a data buffer.

- *Verification of Loop Termination*: This thesis presents a formal method to verify that a loop terminates in HOL4. This methodology is used to verify that a loop within a

BD initialization function is correct, further validating the correctness of the driver.

The first two contributions acheive the first step in proving memory isolation of a DMAC as it pertains to the device driver. Proving that a DMAC configuration preserves memory isolation ensures that the DMAC is incapable of reading or writing memory that was not originally intended to be read or written. This would mean that attacks such as data leaks and overwriting critical kernel code are no longer possible using the DMAC, reducing the likeliness of possible attacks on a system.

7.2 Limitations

The following are some of the limitations of the work done and methodology used in this thesis:

- *Model Validation*: The model design approach used in this thesis must be done manually, therefore there is the possibility of human error. This means that modeling a device driver requires a deep understanding of the inner-workings and interactions of the device, OS, and driver. Although the process of verification may bring existing errors to attention, it is not enough to assert that the model is perfectly representative of the original driver.

- *Time Consumption*: Manually modeling a device driver as well as verifying properties of this model is time consuming, and requires significant experience with ITPs such as HOL4. This means that the approach used in this thesis may not be attractive to companies that do not require complete reassurance that a device driver product provides memory isolation due to the overall commitment to the method.

- *Driver Interactions*: Drivers interact with the OS and the hardware device, so this thesis assumes correctness of both of these systems. However, this may not always be the case, so a model of the device as well as the OS would be needed to complete the verification process. This would add more time and complexity to the system's model.

- *Other Vulnerabilities*: Although the secure operation of a DMAC ensures memory isolation, drivers often include other capabilities as well as DMAC configuration, which could also contain security vulnerabilities. For example, other parts of the driver may contain bugs that are sensitive to buffer overflows, integer overflows, division by zero, null pointer dereferences, etc. [6]. To ensure complete security of a device driver, these vulnerabilities would also need to be checked.

7.3 Future Work

To address the limitations expressed in Section 7.2 regarding model validation, one area of future work would be to synthesize C code from the model that has been created in this thesis. One approach in doing this is outlined by Gawali [12]. Once the C code is generated, then the code could be put in place of the original X550 driver code and tested to ensure that the original operation of the driver is preserved. This would ensure the overall correctness of the driver's model, proving that the verification is true.

Another area of future work is to prove the remaining properties required to verify memory isolation of a device driver. There are three properties that prove memory isolation according to Haglund [14]. These properties are listed below.

1. All BDs must only specify reads and writes to buffer addresses that are in readable and writable memory regions.

2. BDs must exist only in readable and writable memory.

3. BDs must not overlap with addresses that will be written or read by the DMAC.

The first of these properties is proven in this thesis. The second property would be proven in the exact same way the first property is proven, by isolating the functions used to allocate BDs and proving that they are allocated in valid memory regions. The final property would be proven by analyzing the two models created for the first two properties and verifying that the memory regions do not overlap according to the DMAC operation. This would provide full verification of a device driver's memory isolation as it pertains to a DMAC.

Integrating this device driver model with a model of the OS as well as the hardware device would also be a possible area of future work. This would allow for an entire system to be modeled and verified, ensuring total memory isolation as well as being able to prove other properties about complete systems.

Bibliography

[1] Eyad Alkassar and Mark Hillebrand. Formal functional verification of device drivers. volume 5295, pages 225–239, 10 2008. ISBN 978-3-540-87872-8. doi: 10.1007/978-3-540-87873-5_19.

[2] Eyad Alkassar, Mark Hillebrand, Steffen Knapp, Rostislav Rusev, and Sergey Tverdyshev. Formal device and programming model for a serial interface. 01 2007.

[3] Jason Andress. Chapter 12 - application security. In Jason Andress, editor, *The Basics of Information Security (Second Edition)*, pages 189–208. Syngress, Boston, second edition edition, 2014. ISBN 978-0-12-800744-0. doi: https://doi.org/10.1016/B978-0-12-800744-0.00012-9. URL https://www.sciencedirect.com/science/article/pii/B9780128007440000129.

[4] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.*, 40(4):73–85, apr 2006. ISSN 0163-5980. doi: 10.1145/1218063.1217943. URL https://doi.org/10.1145/1218063.1217943.

[5] Hao Chen, Xiongnan (Newman) Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 431–447, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908101. URL https://doi.org/10.1145/2908080.2908101.

[6] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450311793. doi: 10.1145/2103799.2103805. URL https://doi.org/10.1145/2103799.2103805.

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. SOSP '15, page 18–37, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815402. URL https://doi.org/10.1145/2815400.2815402.

[8] Janis Danisevskis, Marta Piekarska, and Jean-Pierre Seifert. Dark side of the shader: Mobile gpu-aided malware delivery. In *Information Security and Cryptology – ICISC 2013: 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, page 483–495, Berlin, Heidelberg, 2014. Springer-Verlag. ISBN 978-3-319-12159-8. doi: 10.1007/978-3-319-12160-4_29. URL https://doi.org/10.1007/978-3-319-12160-4_29.

[9] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 321–332, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320146. doi: 10.1145/2491956.2462184. URL https://doi.org/10.1145/2491956.2462184.

[10] Jianjun Duan. *Formal Verification of Device Drivers in Embedded Systems*. PhD thesis, University of Utah, USA, 2013.

[11] Stéphane Duverger. Linux 2.6 kernel exploits. *Journal in Computer Virology*, 4(1): 39–60, Feb 2008. ISSN 1772-9904. doi: 10.1007/s11416-007-0066-9. URL https://doi.org/10.1007/s11416-007-0066-9.

[12] Aditya Rajendra Gawali. Modeling and synthesis of linux dma device drivers using hol4, April 2024.

[13] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 653–669, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu.

[14] Jonas Haglund and Roberto Guanciale. Formally verified isolation of dma. In *Formal Methods in Computer-Aided Design*, 2022.

[15] *Intel® Ethernet Controller X550 Datasheet*. Intel®, 7 2020. v2.5.

[16] Asim Kadav and Michael Swift. Understanding modern device drivers. *ACM SIGARCH Computer Architecture News*, 40:87–98, 04 2012. doi: 10.1145/2150976.2150987.

[17] Moonzoo Kim, Yunja Choi, Yunho Kim, and Hotae Kim. Formal verification of a flash memory device driver – an experience report. volume 5156, pages 144–159, 08 2008. ISBN 978-3-540-85113-4. doi: 10.1007/978-3-540-85114-1_12.

[18] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of

an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587523. doi: 10.1145/1629575.1629596. URL https://doi.org/10.1145/1629575.1629596.

[19] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 249–262, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340915. doi: 10.1145/2872362.2872379. URL https://doi.org/10.1145/2872362.2872379.

[20] A K M Fazla Mehrab, Ruslan Nikolaev, and Binoy Ravindran. Kite: lightweight critical service domains. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 384–401, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391627. doi: 10.1145/3492321.3519586. URL https://doi.org/10.1145/3492321.3519586.

[21] David Monniaux. Verification of device drivers and intelligent controllers: a case study. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software*, EMSOFT '07, page 30–36, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938251. doi: 10.1145/1289927.1289937. URL https://doi.org/10.1145/1289927.1289937.

[22] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an os kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 252–269, New York, NY, USA, 2017. Association for Comput-

ing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132748. URL https://doi.org/10.1145/3132747.3132748.

[23] Willem Penninckx, Jan Mühlberg, Jan Smans, Bart Jacobs, and Frank Piessens. Sound formal verification of linux's usb bp keyboard driver. pages 210–215, 04 2012. ISBN 978-3-642-28890-6. doi: 10.1007/978-3-642-28891-3_21.

[24] Hendrik Post and Wolfgang Küchlin. Integrated static analysis for linux device driver verification. In *International Conference on Integrated Formal Methods*, 2007. URL https://api.semanticscholar.org/CorpusID:2517338.

[25] Oliver Schwarz and Mads Dam. Formal verification of secure user mode device execution with dma. In *Haifa Verification Conference*, 2014. URL https://api.semanticscholar.org/CorpusID:16766459.

[26] Patrick Stewin and Iurii Bystrov. Understanding dma malware. In Ulrich Flegel, Evangelos Markatos, and William Robertson, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–41, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37300-8.

[27] Wikipedia contributors. Circular buffer — Wikipedia, the free encyclopedia, 2024. URL https://en.wikipedia.org/wiki/Circular_buffer. [Online; accessed 12-April-2024].

[28] Leping Zhang, Yongwang Zhao, and Jianxin Li. A comprehensive specification and verification of the l4 microkernel api. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 217–234, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-57249-4.

Appendices

https://doi.org/10.5281/zenodo.11095667