Software Defined Memory Ownership System

by

Alexander Huang

B.S., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author . . . .
Department of Electrical Engineering and Computer Science
August 13, 2020

Certified by . . . .
Dr. Howard Shrobe
Principal Research Scientist, MIT CSAIL
Thesis Supervisor

Certified by . . . .
Dr. Hamed Okhravi
Senior Staff Member, MIT Lincoln Laboratory
Thesis Supervisor

Certified by . . . .
Dr. Nathan Burow
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Under Secretary of Defense for Research and Engineering.

Software Defined Memory Ownership System

by

Alexander Huang

Submitted to the Department of Electrical Engineering and Computer Science
on August 13, 2020, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Memory corruption bugs account for 70% of existing vulnerabilities. Such bugs proliferate because critical code such as operating systems is still implemented in unsafe languages like C and C++. To address this, memory safe languages such as Rust have been developed. Rust provides memory safety in its default usage by performing static and dynamic checks to ensure code conforms to its safety model. However, these checks may be too restrictive for certain OS code. In these cases, programmers must write unsafe code to escape the safety guarantees. Even for kernels developed in Rust, guaranteeing safety for memory mapped input output (MMIO) device interactions remains a challenge given these interactions necessitate unsafe Rust to access addresses that appear arbitrary to the compiler.

We build the Software Defined Memory Ownership System, or SDMOS, that enforces safe MMIO interactions in the Zero-Kernel Operating System (ZKOS), an operating system written in Rust. SDMOS leverages a tagged architecture to embed semantic metadata with IO memory regions and low-level device driver code in addition to policies that define proper MMIO access at varying levels of granularity. We also implement a pipeline to apply tags at the compiler level, minimizing the amount of manual tagging. Our results show that SDMOS eliminates memory corruption resulting from buggy user space applications and device drivers. The main factor that dictates performance of SDMOS is the total number of rules installed to the tag cache. Our evaluations show that SDMOS's cache load should not exceed the capacity of most cache implementations.

Thesis Supervisor: Dr. Howard Shrobe
Title: Principal Research Scientist, MIT CSAIL

Thesis Supervisor: Dr. Hamed Okhravi
Title: Senior Staff Member, MIT Lincoln Laboratory

Thesis Supervisor: Dr. Nathan Burow
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

I would like to express the utmost gratitude to those who have helped me through every step of the MEng process. Thanks to Dr. Howard Shrobe and Dr. Hamed Okhravi for providing the opportunity to this project as well as crucial guidance. They have created a lab atmosphere that is supportive and collaborative, which has benefited my learning greatly.

Thanks to Dr. Nathan Burow for the numerous technical advice that he has given, without which this project could not have been completed. In addition, his incredible patience and well-thought-out feedback throughout the writing process has been a tremendous help.

Thanks to other mentors at Lincoln Laboratory and colleagues from the lab group for their support and useful pointers: Dr. Samuel Jero, Dr. Bryan Ward, Baltazar Ortiz, Claire Nord, Elijah Rivera, Ashley Kim, Josh Hilke and everyone else. Special thanks to Justin Restivo for giving me a jump start at the early stages of this project and providing guidance at critical moments.

Thank you 媽媽 for inculcating me with the importance of education and encouraging me to pursue whatever inspires me. I would not have the opportunities I do without you and I hope I get to tell you all about them someday. Thank you 光旭舅舅 and 秀運阿姨 for the encouragements and support. Thank you for taking care of 媽媽 when I was so far away, I would not have completed my education without your help. Thank you 爸爸 for supporting me through school. Thank you Diane for helping me through some rough times and believing in me. Special shout out to Sid and David for always having my back. Thanks to Jiji the cat for making quarantine bearable and keeping my stress at bay. Lastly, there are countless others who enabled me to be where I am today. Thank you all!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Memory corruption bugs introduced by unsafe languages serve as a common attack vector that can grant adversaries abilities ranging from crashing programs to executing arbitrary code [36]. C and C++ are popular unsafe systems programming languages that are still widely used due to the performance and flexibility they offer. These unsafe languages are able to achieve performance due to the lack of abstractions that guarantee static and run time memory safety [5]. Similarly, they are flexible in that no restrictions are placed on how programmers can allocate memory or how pointers should be manipulated, leading to undefined behavior when an invalid address is inevitably referenced [47, 16, 48].

Lack of memory safety is especially problematic in operating systems, which serve as the foundation for a secure computing system. Common operating systems such as Windows, Linux, and MacOS alike have core implementation roots in C and C++. These languages' lack of memory safety has led to an abundance of spatial and temporal memory bugs in low level OS code. To highlight the severity of these problems, a recent internal review from Microsoft revealed that 70% of their common vulnerability and exposure reports have to do with memory safety [34]. A search on the Common Vulnerabilities and Exposure (CVE) database reveals that there are over 1200 Linux memory related vulnerabilities as of August 2020 [2]. Device drivers are up to seven times more likely to contain bugs compared to the rest of the kernel [9]. Additionally, the Intel e1000e network adapter bug [12] shows that having code other than the device driver interact with MMIO memory can result in the hardware being destroyed.

Recent efforts to improve memory safety include the use of static bug finders such as sanitizers [43] and fuzzers [35, 40, 11]. However, the main problem is that these techniques cannot guarantee all bugs are identified. Formal analysis [25, 24, 22] can offer more concrete guarantees on code correctness but is extremely time consuming to utilize. This is discussed in more detail in subsection 2.1.1. Device driver security has also been an active area of research. Safe device driver interface generation [33], domain specific device driver languages [10], and reducing driver privilege [39] are all techniques that have varying trade-offs for security and performance.

To address the existing vulnerabilities in operating systems, this thesis describes the design, implementation, and evaluation of the Software Defined Memory Ownership System (SDMOS). In SDMOS, we use Rust [32], a programming language with strong memory safety guarantees provided by its ownership type system and a tagged architecture to prevent memory corruption in unsafe device driver code within the kernel by enforcing a version of Rust's type system.

Rust has two models of code: safe and unsafe. In safe Rust, the compiler prevents programmers from writing code that has memory corruption bugs by performing static and dynamic checks against its safety model. Rust's safety model includes a set of overarching rules that, when followed, provides memory safety. However, these rules reduce flexibility. There exists situations where safe Rust's compiler checks are too restrictive and would prevent programmers from performing the necessary raw memory manipulations required in OS programming. Thus, Rust allows programmers to use the `unsafe` keyword to write code that escapes these checks. This is necessary because there are instances of OS code where no current compiler can be expected to provide safety guarantees.

To reap the security properties of Rust, we build SDMOS on the Zero Kernel Operating System (ZKOS). ZKOS is a fork of Tock [29], an OS written in Rust. ZKOS aims to utilize Rust's memory safety guarantees wherever possible in order to eliminate memory corruption bugs. However, ZKOS depends on modules that have underlying `unsafe` implementations such as device drivers. In ZKOS, physical peripherals' registers are mapped to a range of memory addresses which must be read and written to by device driver code. While this is a simple pointer operation in C/C++, it can only be done with unsafe code in Rust, exposing this part of the kernel to memory corruption bugs. We categorize this interaction as *memory mapped input output*, or MMIO, interactions. Rust providing memory safety guarantees for MMIO interactions would be tantamount to its compiler having semantic understanding of all hardware peripherals in order to ensure legitimate memory accesses. This kind of compiler does not currently exist.

Tagged architectures can address the lack of safety guarantee in unsafe Rust by allowing for semantic information to be embedded at run-time. A tagged architecture associates metadata tags with every word in memory. These tags can encode a multitude of information from access rights to data type that gives context to the data at an address. The processor can then validate tags against a programmable set of policies to determine whether certain instructions should be allowed. We write policies to enforce a concept of *ownership* for MMIO devices, allowing MMIO interactions only when device driver code has the proper ownership of the memory range on which it operates. This provides memory safety for the otherwise unsafe MMIO interaction. Ownership also has the added benefit of safety for hardware devices, which can prevent catastrophic hardware failure as seen in the Intel e1000e bug.

In SDMOS, we enhance the ZKOS tagged architecture policies to improve the security of MMIO interactions. Since ZKOS is a security focused OS written in Rust, we benefit from the safety properties of the language where possible, preventing memory corruption altogether. In MMIO driver code where unsafe Rust is necessary, we leverage tagged architecture policies to limit memory regions with which unsafe code can interact using the concept of ownership. By limiting the scope of memory accessible to unsafe code, we eliminate the possibility of corruption outside those regions. Although unsafe code blocks can still corrupt regions they are authorized to interact with, we can guarantee undefined behavior does not occur.

Our contributions are as follows:

- Extend Rust's safety by defining the notion of ownership for IO memory region.
- Enforce compartmentalization for MMIO functions with tagged architecture policy.
- Design a pipeline for applying tags to unsafe code in device drivers and IO memory ranges at compile time.
- Evaluate cache load as a key performance metric for policies at three levels of protection granularity.

Chapter 2

Background

SDMOS consists of three main components which will be discussed in this section. First, we explore the security benefits as well as limitations of safe and unsafe Rust. Next, we examine Tock, an exiting operating system written in Rust upon which ZKOS is built. Lastly, we give an overview of tagged architecture and example policies.

2.1 Rust

Rust is a programming language designed with two primary goals: performance and safety. Notably, Rust achieves memory safety without the use of garbage collection (GC). We recognize performance and safety have historically been viewed as orthogonal goals since safe languages typically provide memory safety by utilizing GC, whose overhead trades off performance for safety [5]. This has led to a proliferation in languages such as C/C++ in systems development that have little safety guarantee but high performance. In this section, we review some common memory safety problems observed with C/C++. Additionally, we examine the memory safety guarantees Rust provides as well as limitations it imposes on systems programming.

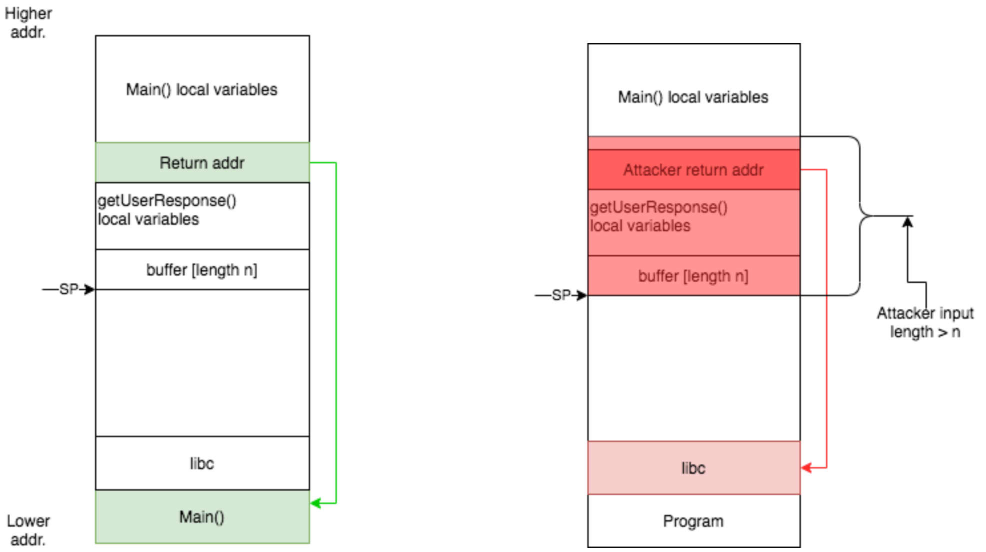
2.1.1 Memory Safety Problems

Without memory safety built into a language, programmers are responsible for ensuring that buffers respect boundaries and all memory locations with which code interacts are valid. Since C and C++ do not have memory safety guarantees, programmers have the ability to specify arbitrary memory locations and de-reference them. This becomes problematic when the specified location is invalid, leading to corruption.

Another problem is the lack of bounds checking, which can allow for an attack vector such as return oriented programming (ROP) [42]. In ROP, attackers inject argument values and overwrite return addresses in order to execute arbitrary code. Suppose a programmer allocates a length n character array `buffer`; nothing prevents code from assigning values beyond that boundary unless explicit checks are performed. A classic example is when the programmer calls `gets`, a function that takes one line of characters from `stdin` and copies it to a specified memory address without checking that the input is less than n. In Figure 2-1a, `getUserResponse()` uses `gets` and puts user input in `buffer`. If the length of input string exceeds n, it can overwrite memory locations that store critical values such as return addresses as seen in Figure 2-1b. In benign cases, this will only result in a segmentation fault. However, an attacker can specify a return address to point to `libc`. Using the right arguments and function from `libc`, the attacker can ultimately execute arbitrary code.

2.1.2 Attempts to Secure C/C++

Current state-of-the-art mitigation for memory corruption include Google's AddressSanitizer [41], referred to as ASan, which consists of compiler and run-time checks to catch memory corruption bugs. However, this tool incurs a significant run-time CPU slowdown of 2x and increases memory usage by up to 3x. Additionally, ASan cannot detect access to uninitialized memory. A wide range of similar techniques for sanitzation [43] or runtimes defenses [47] have also been proposed in the community, which provide various performance and security trade-offs. Another mitigation is kAFL



(a) Ownership of entire MMIO region

(b) Ownership of individual device regions

Figure 2-1: Depiction of ROP attack

[40], which uses a technique known as *fuzzing* to detect bugs in kernel code. Fuzzing randomly mutates test case in order to explore program execution paths, providing substantial code coverage. While this can detect bugs during testing, it is not able to provide any security monitoring for deployed code nor is it guaranteed to identify all existing bugs. Formal verification [25, 24, 22] is another technique which utilizes mathematical modeling and proofs to identify and prevent entire classes of bugs. Although formal verification provides the strongest guarantees for memory safety, it is not currently scalable due to the time consuming nature of producing proofs. Despite these efforts to solve this problem, a majority of bugs today still arise due to the lack of memory safety. Writing programs and OSes in languages with poor memory safety is a fundamental problem that leads to a cat-and-mouse chase in securing exploits after they are found and damages already incurred.

2.1.3 Safe vs. Unsafe Rust

There are two categories of code in Rust: safe and unsafe. Safe Rust guarantees memory safety so programs cannot overflow buffers or de-reference raw pointers. The compiler will check for these violations at compile time as well as inject run-time checks if needed and raise an error upon any of the violations described in subsection 2.1.1. These checks ensure attacks like ROP cannot occur and that all memory locations referenced are valid. However, under safe Rust, a raw pointer operation to arbitrary memory space is not allowed. There are cases in OS code such as interacting with hardware peripherals where pointer operations are necessary. To circumvent the limitations imposed by safe Rust, programmers can write unsafe Rust code.

As its name suggests, unsafe Rust code does not have memory safety guarantees and therefore have the safety and flexibility characteristics of C/C++. Operations like raw pointer access and inline assembly are allowed only in unsafe Rust since ensuring safety for them is not currently feasible at the compiler level. For example, pointer de-references can be used to change a value that has been marked immutable or break safe Rust's single ownership model by generating multiple mutable references. Unsafe code can also cause corruption in safe Rust since unchecked pointer operations can mutate data at any address in the memory space of a given process.

We note that unsafe code is not necessarily buggy. For example, if all the addresses provided to unsafe functions in device drivers are valid, memory corruption will not occur. Problems only arise when an invalid address is provided. Therefore, it is important to define clear specifications about any unsafe code's assumption on input values in order to avoid undefined behavior and facilitate seamless integration with safe Rust. SDMOS will further prevent undefined behavior from propagating by implementing appropriate policies.

2.2 ZKOS

The operating system that SDMOS runs on is ZKOS, which is forked from Tock, an OS written in Rust. In this section, we give an overview of MMIO devices and Tock's security benefits.

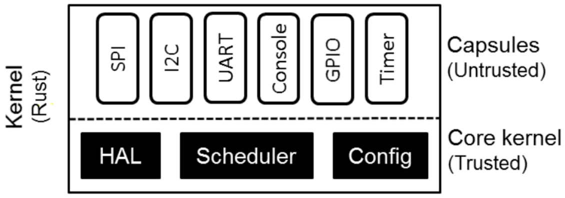


Figure 2-2: Tock's Kernel Structure

2.2.1 Tock

Given the lack of safety guarantees in legacy languages, Tock's Rust implementation is a good starting point for security. Operating systems written in safe languages such as Spin [7] and Singularity [22] have had varying degrees of success but due to dependence on garbage collection, performance has suffered. Since Rust does not utilize garbage collection, Tock does not suffer from the same problem.

Tock's kernel comprises of a trusted core kernel and untrusted capsules as shown in Figure 2-2. To get the most from the safety benefits of Rust, Tock minimizes the amount of unsafe code on which the core kernel depends. Capsules are used to compartmentalize untrusted implementations within the kernel, which includes device drivers. Capsules are Rust structs that can protect their internal state against other capsules by not exporting certain functions or fields. Rust's safety guarantees are applied to capsules. However, device driver capsules must depend on unsafe implementations given that interacting with memory mapped hardware necessitates raw pointer operations, resulting in an attack vector for adversaries. SDMOS aims to eliminate this attack vector.

2.2.2 MMIO Devices

A practical example in which an OS must manipulate raw memory is in device drivers. Hardware devices in computer systems such as GPUs and network adapters have registers mapped to a range of addresses in memory. This is commonly referred to as memory mapped input/output devices, or MMIO for short. A diagram of MMIO space is shown in Figure 2-3

In order for the drivers to access MMIO devices, they must write to and read

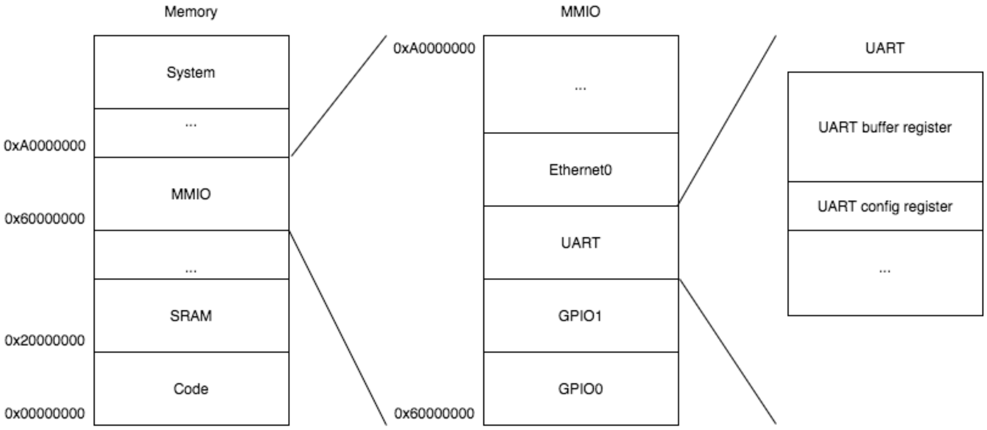


Figure 2-3: Example MMIO memory layout

from these addresses. Doing so is dangerous because correct behavior depends on consistency between the assumed and actual memory layout. Memory layout varies depending on the hardware used. Even within a family of boards, there are a variety of different configurations. Any deviation can result in reading from or writing to critical memory regions that are not reserved for MMIO, leading to undefined behavior. Unfortunately, Rust makes no safety guarantees for raw pointer interactions, leaving device driver code a significant attack surface.

To make the problem worse, many OS implementations treat MMIO memory and regular memory identically even though they are semantically different. Writing to and reading from MMIO space can induce side effects since it can trigger hardware peripheral actions whereas regular memory does not. This is exemplified in the e1000e device driver bug that caused fatal damage to the e1000e network card when a regular memory write was allowed to execute on its mapped memory, causing the network card to be bricked permanently [12]. The inadvertent write occurs when a kernel tracing subroutine, whose job is to replace certain function call instructions at runtime to `nop`s, accidentally writes to the e1000e's memory mapped space, which is illustrated in detail in Figure 2-4. The `cmpexchg` instruction that replaced the function call is supposed to be "safe" since it first checks whether the target memory location actually contains a function call before changing the opcode to a `nop`. The probability that the MMIO region contained the exact opcode that matched the function call instruction

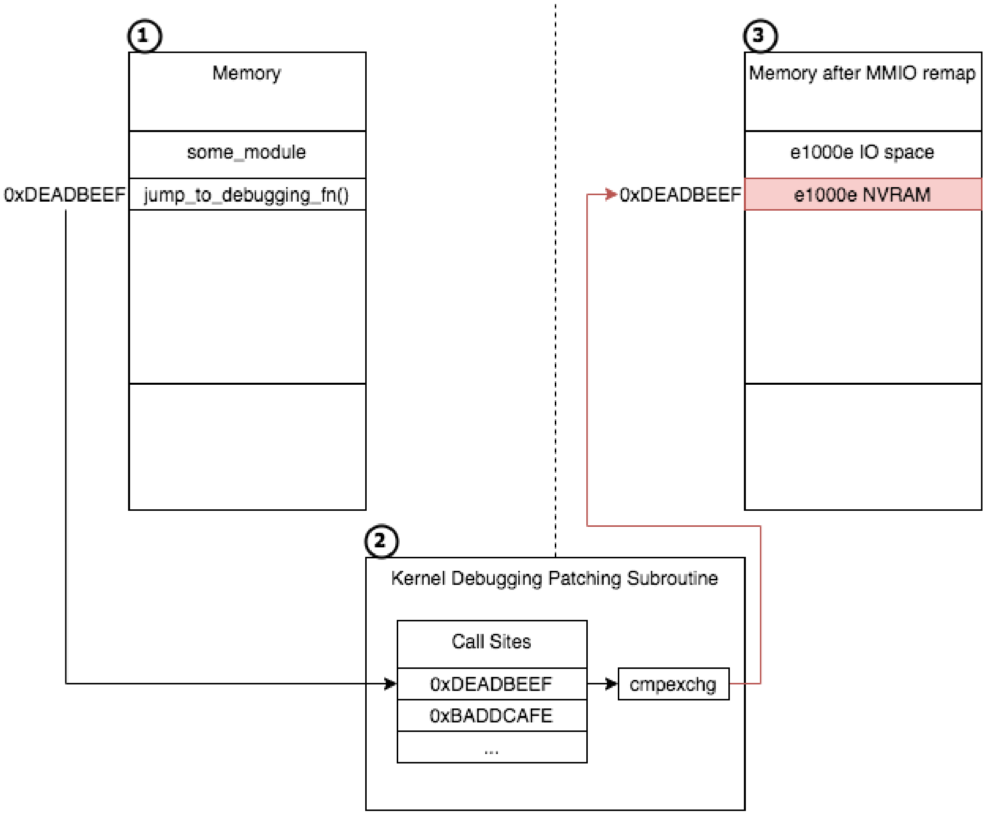


Figure 2-4: The e1000e bug: In 1, the memory is loaded with an arbitrary module. To enable debugging, the compiler adds a function jump at the beginning of the module. In 2, the kernel has a runtime patching subroutine which notes all debugging function call sites and patch the debug function jump if debugging is not enabled. 3 shows a view of the memory after the module is freed and the e1000e is mapped to where the module used to be. Unfortunately, the `cmpexchg` instruction executes since it still believes 0xdeadbeef is the callsite and accidentally writes to the e1000e NVRAM space, destroying the device.

is low enough that this bug might have never occurred. However, `cmpexchg` always performs a write no matter the result of its check (i.e., even if the target memory does not contain a function call, it will rewrite the existing memory value). This results in defined behavior in regular memory but writing to IO regions may cause hardware state to change or may not be permitted at all. In the e1000e's case, it was destroyed due to writes to its NVRAM region. Tock does not distinguish regular memory from IO memory thus its peripherals are susceptible to the same fate as the e1000e network driver. SDMOS will prevent this from happening by compartmentalizing IO memory so only device driver code can interact with it.

2.3 Tagged Architecture

Tagged architectures allow for every word of memory to be augmented with a tag which can be used to give context to the data. With tagged architectures, we can effectively sandbox unsafe Rust code used to facilitate MMIO interactions by tagging the region and writing policies to enforce defined interactions. In this section, we discuss Dover's core tagged architecture [45] implementation components and give an example Read/Write/Execute (RWX) policy to illustrate Dover's domain-specific policy language syntax.

2.3.1 Dover's Implementation

There are a few elements in Dover's implementation relevant to SDMOS. Below, we provide an overview to how tags are checked upon every instruction in addition to the significance of the Policy Execution Core (PEX) and policy rule cache.

Tags

At a high level, tags allow semantic information to be appended to registers and memory at the word granularity. We note that in Dover's implementation, a tag is actually a pointer to metadata, or the actual bytes, that describes the data at a particular address. This indirection is due to the fact that metadata is stored in a location not accessible by the CPU for security reasons. For SDMOS we will refer to tags and metadata interchangeably. We define *tag sites* which are a collection of relevant tags associated with the current instruction. Tags in each of the tags sites must be reviewed prior to the execution of the current instruction. In Dover's implementation there are six possible tag sites and each site is capable of holding multiple tags:

1. `ci`: the current instruction
2. `env`: the program counter register
3. `op1`: the first operand to the current instruction
4. `op2`: the second operand to the current instruction
5. `op3`: the third operand to the current instruction
6. `mem`: the memory referenced

PEX and Rule Cache

The PEX, is central to the operation of Dover's tagged architecture. It is a co-processors that maintains tags on every word of memory and checks these tags against software defined policies to determine if an instruction should be allowed to execute. Policy rules are also only accessible to the PEX and is not modifiable by the application processor's CPU meaning not even the kernel can directly mutate tags or policy rules. Therefore, prior to every instruction execution, the application processor has to query the PEX to see if there are any policy violations. In order to increase performance, a rule cache is implemented within the application processor, which allows policy evaluation results to be cached and fetched without being stalled by the PEX query. The cache's lookup key is formed by matching the combination of tags on each tag site.

2.3.2 Policy

Policies allow us to program the PEX to enforce expected behavior given some combination of tags on each tag site by either allowing an instruction to execute and

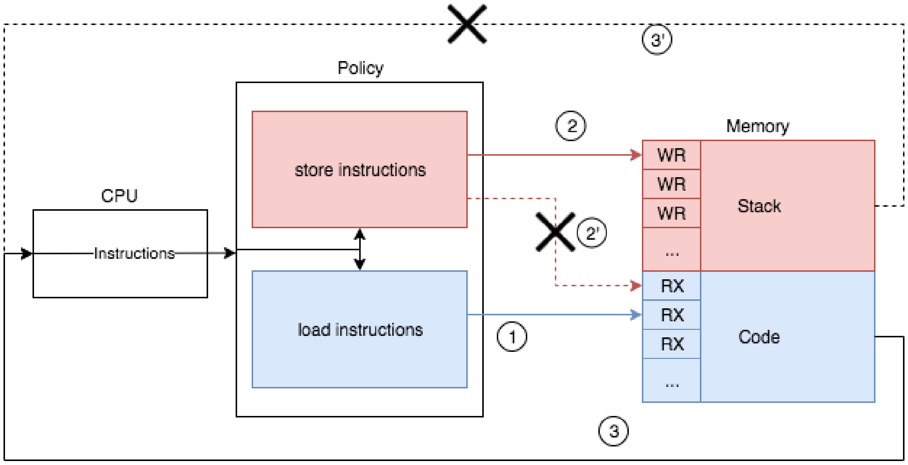


Figure 2-5: Example read, write, execute Policy: 1. load instruction successfully reading from code, which has the read tag 2. store instruction successfully writing to stack, which has the write tag 2'. store instruction failing to write to code 3. code with executable tag gets executed 3'. code from the stack is prevented from executing without the executable tag

update tags in each cache site or triggering an interrupt that stops the application processor from execution.

As an example, we give a high-level overview of how to use the policy language to enforce proper read, write, and execute (RWX) behavior. First, an application binary file containing sections for code and stack is tagged with a combination of read, write, or execute tags on the `mem` cache site. This produces a new ELF file, augmented with tags that can be read by the PEX at boot-time. Next, we write a policy which specifies:

1. all current load instructions must check for the read tag at the desired load address

2. all store instructions must check for the write tag at the desired store address

3. all instructions that are executed must have the executable tag

The PEX will read the tags for all relevant operands and instructions at run-time and upon failure to meet the checks above, crash the program.

2.4 Threat model

Attackers can exploit a variety of attack vectors to compromise security. The kernel code may contain implementation bugs due to its low-level nature and lack of formal verification. Attackers can invoke system calls that may have underlying unsafe implementation to circumvent the safety guarantees of Rust. Given ZKOS has a flat address space that is shared between the kernel and userspace applications, a buggy or malicious userspace application can use this as an attack vector. Attackers can readily access memory space and interact with MMIO devices with pointer operations in userspace applications.

However, there are some limitations for the attacker. We assume the Rust compiler is bug free and thus attackers cannot break the memory and type safety guarantees in Rust code that is not explicitly marked unsafe. Attackers also cannot tamper with Dover's tagged architecture hardware and we assume its implementation is bug free, allowing us to trust the security benefits our tagged architecture provides. Although the attacker can input corrupted values with external hardware peripherals to MMIO memory regions, this should not result in undefined behavior.

Chapter 3

Design

MMIO device drivers must perform raw pointer operations to interact with hardware peripherals, requiring unsafe Rust code. This presents itself as an attack vector since unsafe Rust code does not possess any safety guarantees. In addition, hardware peripherals can be susceptible to damage or data exfiltration if unauthorized code is able to interact with them. To solve these issues, we introduce the notion of *ownership* for each device driver and require any interaction with MMIO memory regions to be contingent on valid ownership. A driver owns the memory region to which its hardware device is mapped as depicted in Figure 3-1. This definition of ownership can also be applied at finer *granularity* (e.g., to functions inside device drivers and specific registers in MMIO regions). Drivers are *compartmentalized*; they can only interact with memory regions they own. It is possible to compartmentalize device drivers due to the predictability in their structure: any raw pointer operation must be applied to a static memory range to which a hardware peripheral has been mapped.

To compartmentalize MMIO drivers, we design policies that ensure all pointer reads and writes to any hardware device must come from the appropriate device driver and respect ownership. We also explore the trade-offs in varying the granularity of ownership. At the coarsest level of granularity, we distinguish ownership between devices and their respective drivers (e.g., UART driver cannot write to GPIO memory range). At the finest level of granularity, we distinguish ownership between registers

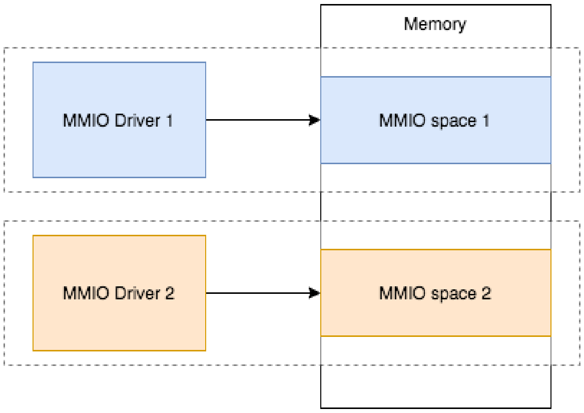


Figure 3-1: Ownership of memory regions by device drivers

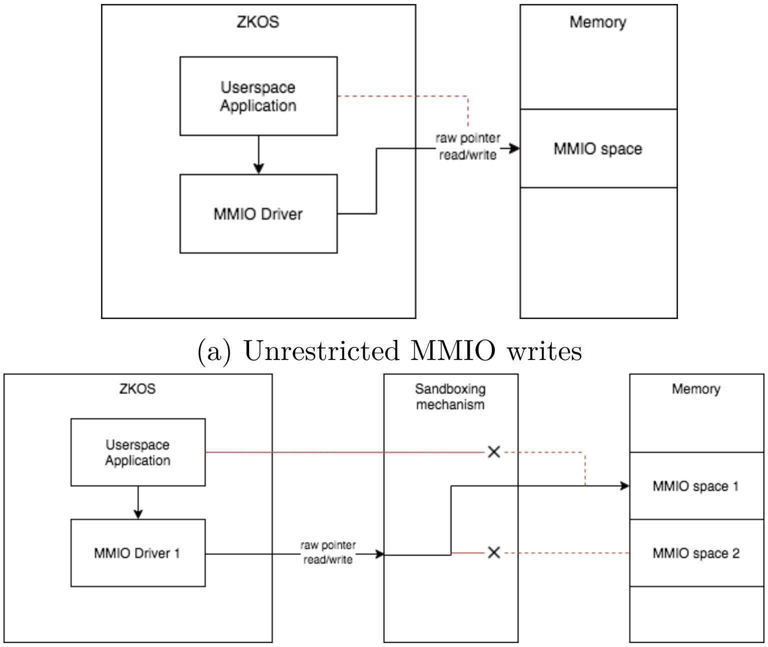
in a device and functions that are allowed to access them within the driver.

Overall, we limit the scope of access for memory regions designated for hardware devices with the notion of ownership and compartmentalization, minimizing the attack surface. In figure 3-2a, user space applications can read and write to the MMIO space with the same privilege as the device driver due to ZKOS's flat address space topography and lack of compartmentalization. Our design introduces a sandbox mechanism as shown in figure 3-2b, which guarantees that any pointer operations to MMIO space come from the corresponding driver and rogue pointer reads and writes will be blocked, compartmentalizing the MMIO driver code in ZKOS. As a practical example, even an exfiltrated pointer to the MMIO space cannot be used by malicious programs since only the appropriate driver can operate on that space. The sandbox mechanism comprises of two main components: a set of tags to be associated with registers, instructions, or memory as well as policies that processes the tags and either allow or disallow an instruction from executing.

3.1 Goals

Goals that guide the design of SDMOS are security, performance, and simplicity.

To maximize security, we follow the principle of least privileges, which states that all components should only be granted the minimum amount of privilege in order to perform their function. Device drivers are only given enough privilege to operate



(a) Unrestricted MMIO writes

(b) Compartmentalized MMIO writes

Figure 3-2: Examples of MMIO writes

on their corresponding memory region. Along with this, compartmentalization will also be employed to establish distinct access privileges between devices and their registers. To achieve this, we note that device drivers need to support read and write operations to the mapped memory range corresponding to their hardware devices. In our application, the mapped memory ranges remain static, allowing us to associate static tags with them at boot time. These static tags enable our tagged architecture to identify memory ranges occupied by hardware peripherals, which allows us to define ownership rules in policies.

We choose to compartmentalize device drivers since all hardware interactions in ZKOS must invoke one of their functions. Thus, these drivers are critical components to secure. Additionally, drivers' ownership relationship to memory ranges of hardware devices are well defined by their specification. Drivers should only read from and write to predetermined locations in memory. Device drivers can be structured to cover all possible register operations with individual function calls to which we can assign privilege, giving us fine control in defining ownership granularity. Due to the static nature of MMIO operations in ZKOS, it is feasible to sandbox them.

To optimize performance, we minimize the number of tags used. Since the bottleneck of the system will be fetching and computing policy rules and loading them in the cache, minimizing the number of tags will ensure that the hit rate is high, eliminating most of the performance overhead.

Lastly to achieve simplicity in implementation, we design the policy to be as small as possible. The scope of each policy should be just enough to cover a single hardware driver and be modular enough to be run independently or in conjunction with other policies. We also consider the amount of manual modifications that ZKOS needs in order to scale with additional devices and aim to minimize this work.

3.2 Policy Design

We use the following categories of tags to give context to memory and enforce compartmentalization:

1. A *property* tag is used to associate unique identity with each MMIO region and distinguish it from regular memory. This type of tag can be applied at varying levels of granularity, ranging from entire device ranges to single registers.

2. An *owner* tag is required to access memory that is tagged as property.

Policies must process the tags defined above and only allow execution for load and store instructions that interact with MMIO space when the owner and property tags match. We design three policies that enforce varying levels of ownership and explore the design trade offs between them. The level 1 policy defines ownership between drivers and all MMIO regions but does not distinguish ownership between different devices. The level 2 policy increases granularity and defines ownership between respective device drivers and the memory range to which their hardware has been mapped. The level 3 policy has the finest ownership granularity, defining it between code within the driver and registers they should access. These policies can be run individually or in conjunction other policies. See Figure 3-3 for an illustration of allowed and disallowed MMIO interactions and Figure 3-4 for an illustration of each policy level and its corresponding memory granularity.

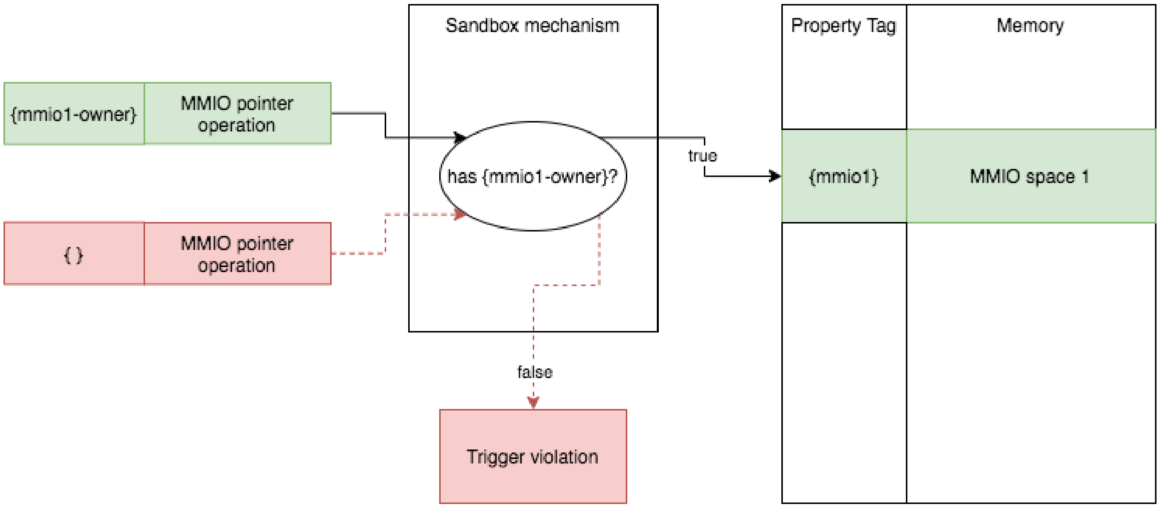
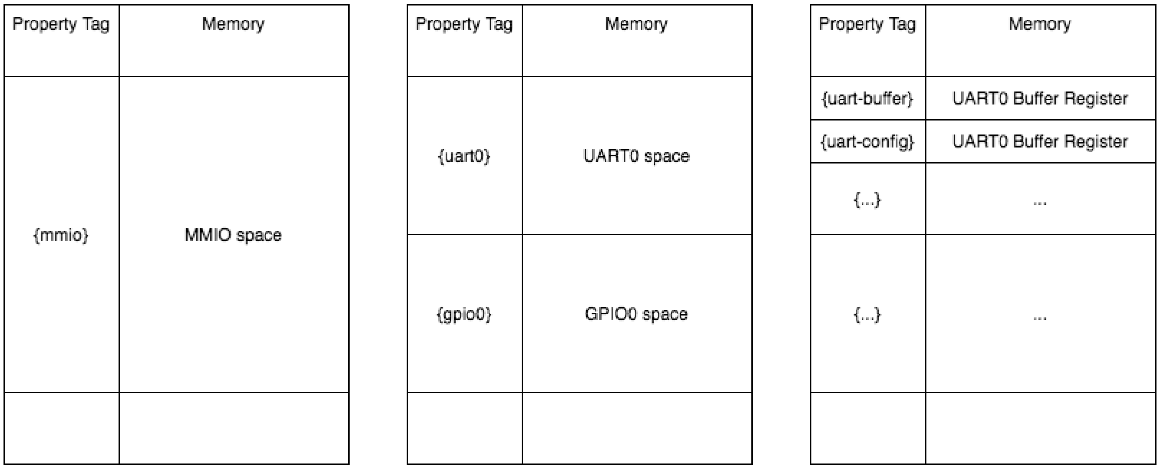


Figure 3-3: Property and owner tags used in sandbox mechanism, tags are shown in braces



Property Tag	Memory
{mmio}	MMIO space

(a) Ownership of entire MMIO region

Property Tag	Memory
{uart0}	UART0 space
{gpio0}	GPIO0 space

(b) Ownership of individual device regions

Property Tag	Memory
{uart-buffer}	UART0 Buffer Register
{uart-config}	UART0 Buffer Register
{...}	...
{...}	...

(c) Ownership of individual registers

Figure 3-4: Available ownership levels, increasing granularity from left to right

3.2.1 Level 1 Policy: MMIO Space Ownership

All MMIO memory spaces will be associated with an `mmio` property tag. Any load and store instructions that interact with memory tagged with `mmio` require an `mmio-owner`

owner tag. The main security goal of the level 1 policy is to elevate the privilege needed to interact with MMIO region. Since there is no distinction between memory ranges within any MMIO region, the level 1 policy only requires one pair of ownership tags for all device drivers. Therefore, little overhead will be incurred because the number of tags required stays constant even with increasing number of hardware devices. This is the simplest and least costly form of compartmentalization in our design but it does not enforce correct access behavior between different devices. For example, any compromised driver may interact with any hardware devices.

3.2.2 Level 2 Policy: Device Driver Ownership

In the level 2 policy, ownership is given based on hardware devices and corresponding drivers. In the previous design, any MMIO driver may interact with any hardware devices. This is undesirable given our goal of least privileges. In this policy, each device driver will own the memory region to which its hardware is mapped and can therefore only interact with that region. For example, the UART device may receive a `uart0` property tag and the GPIO device may receive a `gpio0` property tag. Reads and writes to the memory range of each device will require respective owner tags such as `uart0-owner` and `gpio0-owner`.

Compartmentalizing devices offers improved security given the increased granularity in ownership compared to only compartmentalizing the MMIO region. Even if one device driver is compromised, it can only corrupt memory that it owns. However, this level of ownership incurs a linear increase in the number of tags used with each hardware peripheral, potentially decreasing performance if the total number of tags exceed what can be stored inside the tag cache.

We note the level 2 policy makes no guarantee on compartmentalization within device drivers. This means that any function inside a device driver may access any register so long as the registers are owned by the driver. This is particularly problematic in devices that have more valuable information in certain registers. In order to address this problem, we further increase granularity and define ownership for registers in the level 3 policy.

3.2.3 Level 3 Policy: Register Ownership

Peripherals such as the UART module have varying privilege concerns due to the availability of more sensitive information in certain registers. For example, the UART module's buffer register may be more attractive to an attacker given that it communicates with other devices and transfers important information or it serves as a gateway to gain shell access and therefore control of the system. Thus, it is advantageous to define ownership at the register level. In this particular case, we draw a distinction between the configuration registers and the buffer register by tagging the two groups of registers with the `uart-config` and `uart-buffer` property tags. In order to access each register, the appropriate owner tags must be present.

Compartmentalizing registers offers the most security and least possible privilege. It is also the most expensive in terms of the tags used since each device typically has a number of registers. Note that in the design example above, it is possible to assign a unique owner tag to each register, but registers are grouped together in order to reduce tag usage. The number of tags used will scale linearly with the number of registers in each device in the level 3 policy. Although this ownership level is the most tag intensive of our three designs, case specific optimizations such as binning multiple registers into the same ownership group can reduce the cost while providing improved security over the level 1 and level 2 policies.

Chapter 4

Implementation

The two major components we implement for SDMOS are a tagging pipeline and a set of security policies. We have chosen to implement SDMOS on Dover's tagged architecture since it fulfills our need for a robust policy language with register and memory tagging capability. The architecture is also flexible enough that it can be integrated with a custom Rust compiler, modified to make tagging feasible at scale. In this section, we outline the initial approach that we took in our tagging pipeline and discuss implementation details of the two components.

4.1 Tagging

There are two classes of tags that must be applied in SDMOS: property tags and owner tags. Applying property tags to memory is immediately supported by the Dover toolchain. At the time of writing, Dover allows tags to be specified across MMIO regions at the device level with a YAML configuration file. This configuration file binds device names to their mapped memory ranges and specify the tags that should be applied. In order to achieve finer granularity we can modify this file to bind register names to respective memory ranges, allowing byte level references.

The application of owner tags differs from that of property tags since owner tags must be applied to code in ZKOS rather than a static memory range. This leads to more complexity as addresses at which MMIO driver functions reside may change if the code base changes. Thus, manual tagging would not scale well with granularity or increased number of hardware peripherals. We implement compiler support to encode owner tags in the ZKOS binary to solve this issue.

4.1.1 Initial approach: Instruction Tagging

Our initial approach assigns owner tags to the load and store instructions within hardware driver functions that interact with MMIO space. This method allows us to perfectly abide by the principle of least privileges since only targeted, relevant instructions acquire ownership.

However, because of the way device drivers in ZKOS were written, all raw pointer operations are made through a function call to a register interface crate as depicted in Figure 4-1. Tagging the instructions within the register interface would only allow us to support one owner since the load and store instructions interacting with MMIO space is called and therefore shared by multiple drivers. This makes it impossible to distinguish or scale to multiple devices by assigning ownership at the register interface level. We considered refactoring the driver code to forgo the register interface and instead add unsafe Rust code to directly write to the MMIO region in each device driver function. This would ensure a load or store operation in each device driver function that we could tag but increase overall complexity. Furthermore, it would

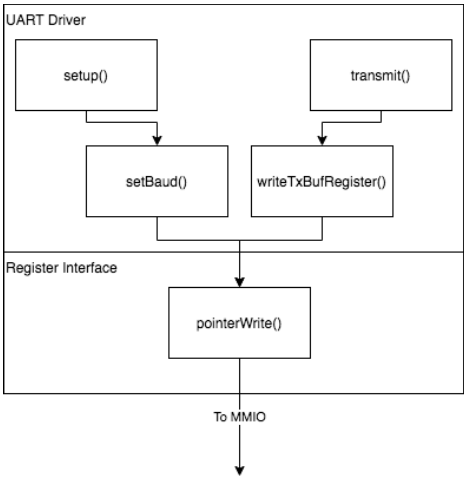


Figure 4-1: Call stack of a UART driver in ZKOS

introduce extensive unsafe code in all drivers rather than compartmentalizing it to one module, decreasing security.

4.1.2 Function Entry and Exit Tagging

Tag	Code
	#fn txdata_write(data)
uart_entry	{ lw a0,0(a0)
...	...
uart_exit	ret }

Figure 4-2: Tags applied at the entry and exit point of an MMIO function

Instead of refactoring load and store instructions into each driver, we choose to tag, and thus give ownership, to individual function in device drivers. This approach still increases the attack surface since instructions that do not necessarily require ownership privilege will have it within the scope of a tagged function. However, implementing ownership at the function level allows us to preserve the register interface, avoiding major refactors to the device driver, yet still enforce compartmentalization with fine grained control over granularity.

To support function tagging in SDMOS and all interactions with a hardware peripheral, device drivers must implement at least one function per register whose sole function is to read to or write from that register. These functions should be as simple as possible, typically containing just a call to the register interface. We define these functions as *MMIO functions* and we apply ownership to all such functions within a device driver. This allows all load and store instructions within the scope of the function, including those resulting from a call to the register interface, to interact with the corresponding property memory region. MMIO functions can have the same privilege as seen in the level 1 policy, or unique privileges as seen in the level 3 policy, allowing variable granularity.

However, we cannot directly apply an owner tag to a function since Dover's tooling only supports tagging of instructions [1]. We also cannot simply tag all instructions inside the scope of a function since ownership needs to be "passed" via function calls to the register interface.To implement function tagging, we introduce two additional tag classes.

1. *entry* tags: These tags are applied to the start of all MMIO functions.

2. *exit* tags: These tags are applied to the end of all MMIO functions.

Entry and exits tags are used to denote the scope of functions for the PEX. An example of entry and exit tags for an MMIO function is shown in Figure 4-2. Upon entering an MMIO function we append an owner tag to the current `env` tag site. The same owner tags will be removed upon exiting the function. This allows SDMOS to keep track of ownership as a state and persist it through function calls. In this way, the actual store and load instruction in the register interface can only inherit the ownership from the particular function that calls them.

4.1.3 Compiler Support

Since SDMOS requires a large number of entry and exit tags to be applied for each hardware device driver, doing so manually would not scale. Instead, we implement logic in the compiler to encode addresses where tags should be applied in the resulting binary. We modify the LLVM backend to check every compiled function against a list of sets of keywords that identify MMIO functions. If the function name string matches all the keywords in a set, an entry tag is applied to the first instruction of that function and an exit tag is applied to the return instruction. We pursue partial function name matching since the LLVM Rust backend mangles original function names, making it difficult to match original function names in a deterministic manner. Refer to Figure 4-3 for a depiction of how the ZKOS binary is tagged.

To avoid having to modify the compiler every time we introduce a new tag or function, we implement a new utility class in the `RISCVAsmPrinter` module, which

[1] Memory ranges can also be tagged but there is no way for the tool chain to calculate a memory range from a function.

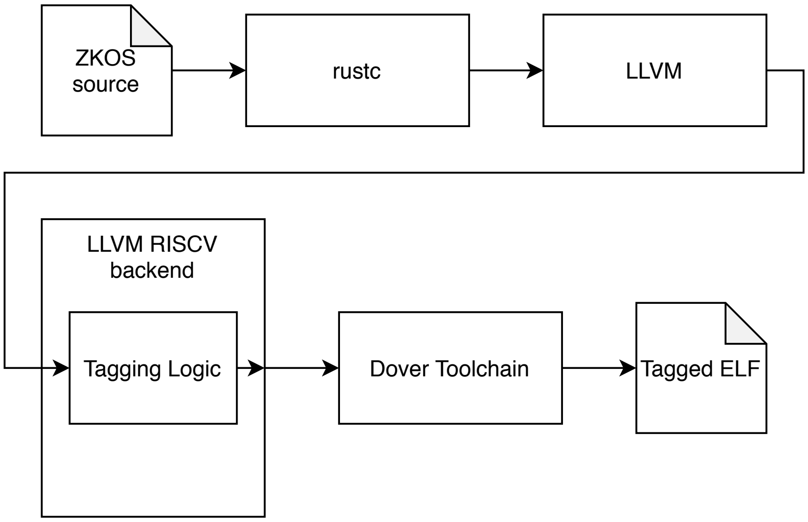


Figure 4-3: Flow of input program to tagged binary

loads a CSV file containing sets of keywords that identify an MMIO function along with the entry, and exit tags that should be associated with that function. This CSV file allows us to dynamically tag new MMIO functions and change granularity without waiting for LLVM to recompile, simplifying the tagging pipeline.

4.1.4 Driver Refactor

In addition to compiler support, we also refactor the device driver code to make calls to MMIO functions rather than directly writing to multiple distinct registers within a function. This allows the compiler support we implement to assign least privilege to MMIO functions. The refactor does not change the overall structure of the driver and the principle can be applied to any new driver code.

In general, MMIO functions should be implemented to the following specification:

1. Interact with only one register: reading and writing to multiple registers would not allow SDMOS to assign least privilege.

2. Minimize function calls: this localizes permission as much as possible.

3. Have a `return` type instruction demarcating the end of a function. In ZKOS, the Rust flag to do this is `#[inline(never)]`.

4.2 Policy

We note the mechanism by which ownership is assigned to each MMIO function as well as the rule that enforces compartmentalization. Additionally, we discuss the limitations introduced with assigning ownership at the function level.

4.2.1 Ownership Assignment

All policies have rules that assign owner tags to the `env` register based on entry and exit tags.

```
allGrp(code == [+fn_entry], env == _ -> env = env[+owner])
allGrp(code == [+fn_exit], env == [+owner] -> env = env[-owner])
```

These two rules check every instruction, denoted by the `allGrp` opgroup, as they execute. Upon finding an instruction that is tagged with the `fn_entry` tag, the first rule dictates an `owner` tag to be appended to the `env` tag site since this indicates the instruction belongs to the beginning of an MMIO function. Similarly, the second rule states that upon exiting an MMIO function, the owner tag should be removed.

4.2.2 Compartmentalization

All policies have rules that enforce compartmentalization. In other words, only the owner of a memory region with a property tag can interact with it.

```
storeGrp (env == [-owner], mem == [+property] -> fail)
loadGrp (env == [-owner], mem == [+property] -> fail)
```

The two classes of instructions that interact with memory are stores and loads as denoted by the opgroup. The rules above ensure that any interaction with memory that has been tagged with the `property` tag without the proper `owner` tag will result in failure.

4.2.3 Limitations

We note that tagging the `env` tag site presents a security limitation regarding interrupt handling. Suppose an interrupt is triggered while in the scope of an MMIO function, code inside the interrupt handler can potentially gain unauthorized access to the MMIO region. However, ZKOS handles interrupts by silencing them and they are only ever serviced upon return from kernel space to user space. Therefore, only the small amount of code in the interrupt handler can illegitimately gain an owner tag. Given the size of the interrupt handler it is feasible to manually verify that it does not interact with MMIO regions. Another potential solution is to introduce an additional policy that pushes any tags on the `env` tag site upon entry to the handler and pops the tags upon exit.

Chapter 5

Evaluation

We evaluate SDMOS with a number of microbenchmarks to highlight the security properties it provides. Additionally, we measure cache load as a key performance metric in each of the policies. The evaluation is carried out on an emulated version of the hardware. Key software versions and machine details are as follows – OS: Ubuntu 18.04.3 LTS, CPU: 32x Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz, Memory: 64 GB, Rustc version: 1.40.0-dev, LLVM version: 9.0.0.

5.1 Security

MMIO operations are fundamentally different from regular stores and loads since they change the state of peripheral hardware. Given that storing out of specification values can have catastrophic consequences as seen in the e1000e network driver bug [12], SDMOS improves security by elevating the privilege required to interact with MMIO regions through ownership. We present microbenchmarks that demonstrate certain security properties. These properties prevent code without the appropriate ownership privileges from inadvertently writing erroneous values to hardware devices. This system would have prevented the e1000e bug from occurring, or at the very least drastically reduced the time it took to find the source of the bug.

The microbenchmarks below illustrate illegal MMIO operations that can be prevented with SDMOS. One microbenchmark illustrates user space applications accessing MMIO space with pointer operations, especially problematic in systems without address space privilege separation such as ZKOS. The other microbenchmarks test cases in which parts of the device driver or kernel have implementation bugs and erroneously interact with the MMIO space, analogous to the e1000e bug.

5.1.1 Userspace access

```c
int main(void) {
  char *uart_ptr = 0x10013000;
  char sensitive_val = *uart_ptr;
  *uart_ptr = ’a’;
  return 0;
}
```

Listing 5.1: User space code accessing MMIO space

Since ZKOS has a flat address space, with every process sharing one address space, it is trivial for malicious programs to access sensitive data that can reside in peripherals such as the UART data buffer. Without SDMOS running, a user space application can simply de-reference a pointer pointing to the UART data register to access the raw bytes. Similarly, it can also trigger writes by the UART peripheral by simply writing values to the data register. Refer to Listing 5.1 for a sample user space C application interacting with a UART device.

SDMOS prevents this from occurring since user space applications are not granted any ownership privilege and are therefore not allowed to directly interact with any MMIO regions. With SDMOS, user space apps are enforced to call ZKOS’ respective MMIO drivers in order to perform any MMIO operations. Running the sample benchmark, SDMOS produces an explicit failure and halts the program execution at the first pointer de-reference operation.

5.1.2 Compromised Driver

Given that device drivers require unsafe code and may have implementation bugs and configuration issues, they can inadvertently interact with an incorrect device. This can be prevented with SDMOS's enforcement of ownership where each device driver can only interact with the memory they own. As seen in Listing 5.2, the UART driver's base address, which should be 0x10013000, is mis-configured as 0x10012000. Running this with SDMOS produces an explicit failure and halts program execution when the UART driver attempts to write to 0x10012000, memory space that it does not own.

```
const UART0_BASE: StaticRef<UartRegisters> =
    unsafe { StaticRef::new(0x1001_2000 as *const UartRegisters) };
```

Listing 5.2: Configuration error in Driver

5.2 Performance

Computing the result from rule policies accounts for a large portion of the performance overhead so we measure the cache load of each policy as the metric of performance. We collect cache load statistics from a QEMU based emulator of the Dover tagged hardware, which is still under active developement. Although the emulated hardware does not give accurate cycle counts, cache performance will determine how many additional cycles are introduced, so ensuring that the miss rate is as low as possible guarantees good performance. Our evaluation provides insight into how much cache space is required in order to fully accommodate the rules in each policy.

We run a user space program that prints to the UART 20 times and toggles a GPIO pin 20 times with three policies that vary in granularity, from level 1 to level 3 as described in section 3.2. We increase the cache size that is used to run these policies until saturation. For the purposes of this evaluation, saturation is defined when the miss rate is minimized and account for misses resulting from initially loading a rule into the cache. The rule cache lookup key is constructed with an array of six sets

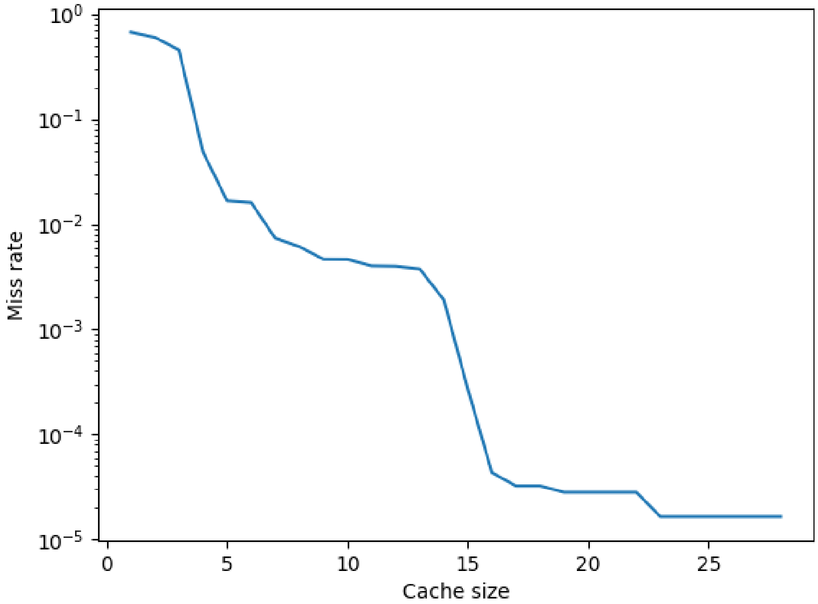


Figure 5-1: Miss rate (log scale) vs. cache size at level 1 ownership

of tags, with each set corresponding to tags that can be applied to the environment (`env`), current instruction (`ci`), operands 1 - 3 (`op1`, `op2`, `op3`), and memory (`mem`), tag sites. We use this fact to perform a worst case analysis for slots required for each policy by calculating all possible lookup keys with (5.1) where $tags_i$ corresponds to the number of unique tags that can appear in each of the six tag sites (`env`, `ci`, `op1`, `op2`, `op3`, `mem`).

$$\prod_{i=0}^{5} \sum_{k=0}^{tags_i} \binom{tags_i}{k} = \prod_{i=0}^{5} 2^{tags_i} = 2^{\sum_{i=0}^{5} tags_i} \tag{5.1}$$

We name this worst case figure the *potential complexity* as in practice, many of the tag combinations that are accounted for should not be observed. Additionally, tag combinations that result in a failure state halts the program execution and therefore do not end up growing the cache size.

5.2.1 Level 1 Ownership

At the coarsest ownership granularity, we expect the lightest cache load given the least number of tags used. Table 5.1 shows the number of unique tags available to

Tag site	env	ci	op1	op2	op3	mem
Unique tags	1	4	0	0	0	1

Table 5.1: Number of unique tags available to each tag site for a level 1 policy

each tag site. The potential complexity with this number of tags would be $2^6 = 64$ slots. In practice, 23 slots are sufficient in accommodating all the rules in this policy. Refer to Figure 5-1 for a graph that shows the miss rate decreasing with increasing cache size. At more than 23 slots, the miss rate stabilizes and only account for the initial misses before the rules were loaded into the cache.

5.2.2 Level 2 Ownership

Tag site	env	ci	op1	op2	op3	mem
Unique tags	1	4	0	0	0	1

Table 5.2: Number of unique tags available to each tag site for a level 2 policy

Structurally, the GPIO and UART policies at level 2 are identical, using the same number of tags as shown in 5.2. The worst case cache load with this number of tags would be $2^6 = 64$ slots. Empirically, the GPIO and UART policy both require 25 cache slots in order to accommodate all their rules. Running both the GPIO and UART policies in combination requires 33 slots. Note that this is less than the sum of cache slots required for running both policies individually since there is overlap in cache rules. For example, both policies share the rule:

```
allGrp(code == _ , env == _ -> env = env)
```

Since this rule matches all instruction regardless of tags, all rules residing in the cache such as those shown in Table 5.3 would overlap across both policies. Rules involving the `gpio` and `uart` owner and property tags would require their own entries in the cache and do not overlap.

	Rule 1	Rule 2	Rule 3
env	{}	{}	{}
co	{allGrp}	{loadGrp, allGrp}	{storeGrp, allGrp}
op1	{}	{}	{}
op2	{}	{}	{}
op3	{}	{}	{}
mem	{}	{}	{}

Table 5.3: Examples of cache keys shared by the level 2 GPIO and UART policies. Rule 1 matches all instructions, Rule 2 matches load instructions, and Rule 3 matches store instructions.

Refer to Figure 5-2 for a graph showing the effect of increasing cache size on the miss rate. As we expect, the miss rate monotonically decreases as cache size increases. Fewer evictions occur until the cache is big enough to accommodate all rules.

5.2.3 Level 3 Ownership

Tag site	env	ci	op1	op2	op3	mem
Unique tags	2	6	0	0	0	2

Table 5.4: Number of unique tags available to each tag site for a level 3 policy

At the finest ownership granularity, we expect the highest cache load given the most number of tags used. For the two register group ownership model we implement in the UART policy, the potential complexity is $2^{10} = 1024$ calculated with the unique tags presented in Table 5.4. Empirically 35 slots are needed in order to accommodate all the rules. Refer to Figure 5-3 for a graph showing the effect of increasing cache size on miss rate. As observed in the previous policies, the experimental cache slot requirement differs from the theoretical potential complexity. Due to the fact that some tag combinations accounted for in the theoretical figure is not observed, the exponential increase in potential complexity is irrelevant. Another way of explaining this observation is that since the level 3 policy still enforces the same ownership mechanism as in level 1 and 2, therefore tag usage only increases linearly due to the linear increase in ownership definitions. The same set of tags can be used by policies to enforce more complicated behavior, which would result in a cache slot requirement

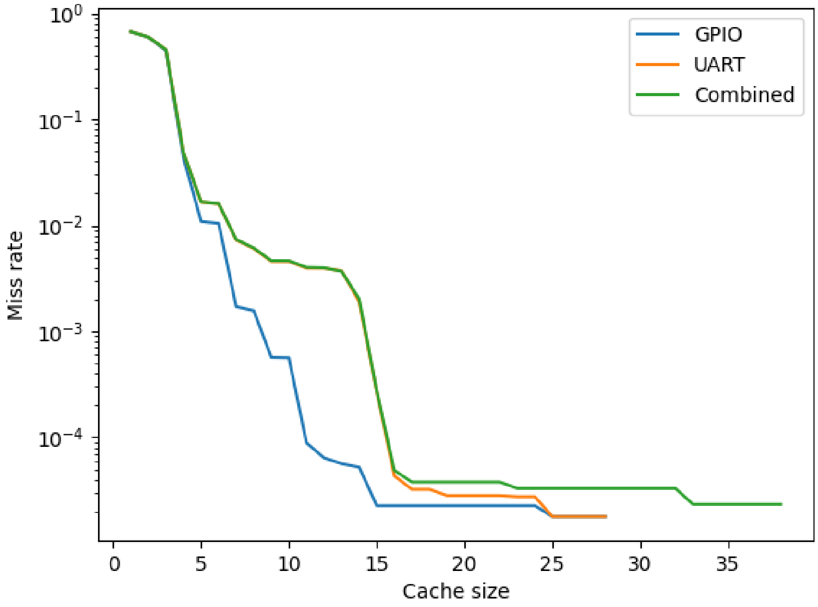


Figure 5-2: Miss rate (log scale) vs. cache size at level 2 ownership with policies running in isolation and combined

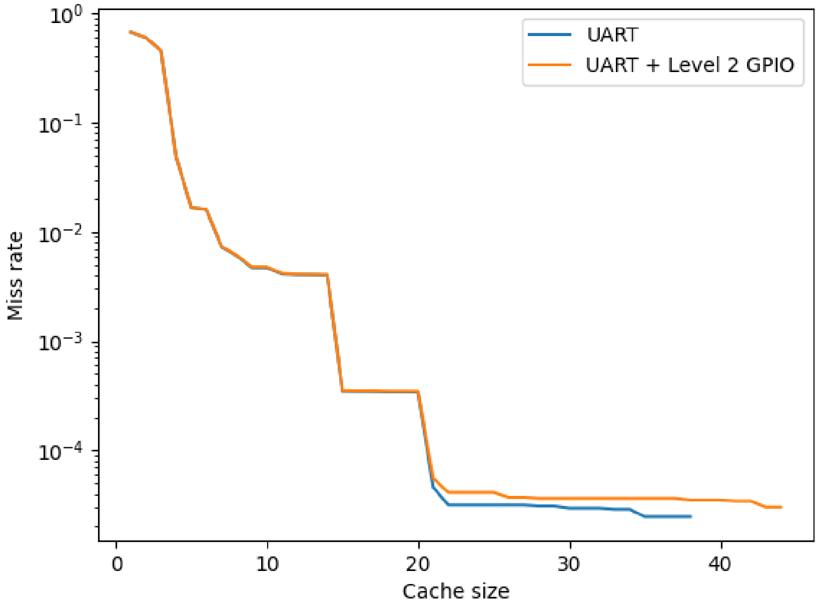


Figure 5-3: Miss rate (log scale) vs. cache size at level 3 ownership with the UART Policy running in isolation

closer to the potential complexity figure. We also note running the level 3 UART policy with the level 2 GPIO policy resulted in an observed cache load of 43 slots. Consistent with the observation made while evaluating the level 2 policy, the cache slot requirement for the combination of two policies is less than the sum of their individual requirements due to overlap.

Chapter 6

Future Work

In this section, we discuss avenues for future research to extend SDMOS. Future work includes supporting multiple hardware device instances, expanding ownership to dynamically allocated memory, providing device driver fault isolation, and designing a peripheral description language that can automatically generate policies.

6.1 Device Instances

Currently, SDMOS does not support applying ownership rules to multiple instances of the same device type. Since we grant ownership at the function level for each device driver, SDMOS does not have a way of distinguishing between device driver instances. Instance support can be implemented by tagging the struct containing the base pointer to a peripheral that device driver initialization creates with a new class of tags named *instance* tags. Then, we can write a policy that uses instance tags to check for device drivers' instance identity, making our definition of ownership stricter. This instance checking policy can be run in conjunction with the current policies.

6.1.1 Dynamic Memory Ownership

SDMOS as presented in this thesis has provided its ownership bindings for static, IO memory. It is possible to broaden the ownership concept to other classes of unsafe code such as context switching. In context switching, processor state is saved to the stack. SDMOS can compartmentalize this chunk of memory in the stack by dynamically tagging it and only allowing interaction by the context switching code.

6.2 Fault Isolation

SDMOS requires ownership in order for driver code to write to IO memory. However, it currently does not prevent buggy device drivers from corrupting other memory regions. To provide fault isolation to device drivers, we recognize that the load and store instructions that perform the pointer write to IO memory must be tagged. We can then write a policy to prevent these instructions from interacting with non-IO memory. The main challenge for this improvement is isolating the correct load and store instructions given that function prologues and epilogues can introduce them.

6.3 Peripheral Description Language

Each device that SDMOS supports requires a distinct policy to be written. A substantial amount of manual work is needed to write these policies although they generally have the same structure, differing mainly in tags used and granularity at which tags are applied. In addition, tags must be defined at several independent stages in the pipeline (e.g. LLVM stage [27] and Dover tool chain stage) and remain consistent. SDMOS would benefit from a peripheral description language that automates the above processes by taking a memory map of its registers as well as functions that should be allowed to access each register then generating device policies and inserting tag definitions in the pipeline. This can greatly reduce the margin for human error and allow reuse of peripheral descriptions for devices with similar topology.

Chapter 7

Related Work

In this section we discuss related work on efforts to increase memory and operating system safety. We also examine alternate methods to specifically provide IO safety guarantees.

Bug Finding. Fuzzing is a technique that randomly mutates inputs to a program in order to test large numbers of program execution paths and provide adequate bug finding coverage. There exists a variety of application specific fuzzers such as kAFL [40], DIFUZE [11] and Syzkaller [17], which specialize in fuzzing kernels JANUS [50] takes this further and fuzzes file system implementations within an OS. Another runtime bug finding tool is a sanitizer, which instrument tests by embedding what is known as *inline reference monitors* (IRMs) into the program at compile time [43]. These IRMs monitor instructions such as memory stores and loads that can potentially lead to a vulnerability. LLVM TySan [14], UBSan [15], MemorySan [44], and EffectiveSan [18] are all examples of memory safety sanitizers that instrument IRMs at the language or IR level. Configuring a sanitizer's bug finding policies is difficult as too strict a policy may result in numerous false positives that require manual confirmation. Since fuzzing and sanitizing both rely on brute-force exploration of a large number of execution paths, it is difficult for them to scale to large code bases. In addition, neither technique can guarantee finding all existing bugs. An interesting work that combines the two techniques is ParmeSan [37], where the fuzzers are directed to trigger sanitizer IRM checks, lowering the time-to-exposure of bugs.

Proving Correctness. Since exhaustively finding all existing bugs in a system may be infeasible, formal verification can be employed to prove the correctness and memory safety of a code base. The Singularity [22] OS is mostly written in Sing# [19], a memory safe language based on C# that can readily be formally verified. However, due to the difficulty in modeling programs for formal verification, it often is generally not practical to apply this technique to large code bases. SeL4 [25], which is a formally verified kernel code base, took over 30 person years to verify.

Safe Languages. Language safety is another attractive option for OS security. BiscuitOS [13] and CliveOS [6] are written in Golang, which provies memory safety. However, they both suffer reduced performance due to overhead incurred by garbage collection. Tock [29] and RedoxOS [4] are both OSes written in Rust, which does not utilize garbage collection to provide its safety guarantees. Along with this, Rustbelt [24] is a semantic model that reveals verification conditions in order to prove the safety of `unsafe` code.

Safe Interfaces. Given device driver code is 7x more likely than the rest of kernel code to be buggy [49] due to their low level implementation, IO memory safety is of paramount importance. SUD [8], SafeDrive [51], DeCaf [39], and Leslie [28] all suggest radical ways of writing driver code centered around safety. In particular, We note the idea of writing safe interface for interaction between kernels and device drivers as an orthogonal approach to our work. Devil [33] is a high level interface definition language used to write specifications for hardware that describe the device and generate safe driver code. Since Devil compiles to C, performance is preserved. Unlike the device specific interface of Devil, Nooks [46] implements a safe interface layer for secure interaction between all device drivers and the kernel. It also keeps track of and validates all kernel data structure modifications made by device drivers, allowing it to facilitate automatic recovery from driver bugs that would have otherwise caused the kernel to crash. The reference validation mechanism [49] (RVM) has features of both Nooks and Devil, supporting a device safety specification language that describes valid hardware interaction per device as well as interposing a global reference validation mechanism interface in the kernel that validates all hardware interactions.

In addition to enforcing memory safety for device drivers, RVM also monitors and rate limits interrupts that device drivers can handle, preventing processor starvation.

Hardware Protection. Complementing the secure interface work, Input/Output Memory Management Units (IOMMU) are a hardware protection mechanism that have also been adopted by major chip manufacturers like Intel [3] and AMD [1]. IOMMUs protect against direct memory access (DMA) attacks in which device drivers act maliciously with DMA enabled devices to access kernel memory ranges, bypassing any MMU protection. There has also been efforts to virtualize the IOMMU [21] so it is available in virtualized guest OS instances. However, IOMMUs have translation and management overhead, which means native DMA performance will be degraded [38, 31, 20]. Additionally, recent survey has shown that IOMMU resistant DMA attacks have surfaced for peripherals using widely adopted standards such as Thunderbolt [30, 26, 23].

Chapter 8

Conclusion

Programming languages like Rust can eliminate entire classes of memory safety bugs. However, current compilers are not advanced enough to provide guarantees for unsafe code, which is necessary to implement parts of an OS. We note that device drivers are at a higher risk of containing bugs due to their low level implementation and often convoluted specifications. SDMOS demonstrates that tagged architecture and policies can be used to extend the concept of memory type safety to unsafe device driver code through the notion of ownership. Our security policies can be fine tuned depending on the threat profile of certain hardware devices and require minimal refactoring of existing driver code. In addition, the tagging pipeline that we have implemented for SDMOS allows tags to be embedded in large, complex code bases such as ZKOS with minimal manual work. Our evaluation also show it is feasible to implement the ownership policies without incurring significant cache load, predicting good performance on hardware. In conclusion, SDMOS offers a novel, software defined solution to ensure memory safety in unsafe device driver Rust code, improving security of ZKOS while maintaining simplicity and performance.

Bibliography

[1] AMD I/O virtualization technology (IOMMU) specification 48882. https://www.amd.com/en/support/tech-docs/. Accessed: 2020-08-11.

[2] CVE search results. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=linux+memory. Accessed: 2020-08-11.

[3] Intel(R) virtualization technology for directed I/O architecture specification. https://software.intel.com/sites/default/files/managed/c5/15/vt-directed-io-spec.pdf.

[4] Redox os. https://www.redox-os.org/. Accessed: 2020-08-11.

[5] Abhiram Balasubramanian, Marek S Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamarić, and Leonid Ryzhyk. System programming in rust: Beyond safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 156–161, 2017.

[6] Francisco J Ballesteros. The clive operating system. In *Technical report, Technical report*. 2014.

[7] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[8] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *USENIX annual technical conference*. Boston, 2010.

[9] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, 2001.

[10] Christopher L Conway and Stephen A Edwards. NDL: a domain-specific language for device drivers. *ACM Sigplan Notices*, 39(7):30–36, 2004.

[11] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2017.

[12] John Criswell, Nicolas Geoffray, and Vikram S Adve. Memory safety for low-level software/hardware interactions. In *USENIX Security Symposium*, pages 83–100, 2009.

[13] Cody Cutler, M Frans Kaashoek, and Robert T Morris. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 89–105, 2018.

[14] LLVM Developers. Tysan: A type sanitizer, 2017.

[15] LLVM Developers. Undefined behavior sanitizer, 2017.

[16] Mark E Donaldson. Inside the buffer overflow attack: Mechanism, method & prevention. *GSEC Version*, 1(3):5, 2002.

[17] David Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.

[18] Gregory J Duck and Roland HC Yap. Effectivesan: type and memory error detection using dynamically typed C/C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–195, 2018.

[19] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, 2006.

[20] Brett Ferdosi Gutstein. *Towards Efficient and Effective IOMMU-based Protection from DMA Attacks.* PhD thesis, 2018.

[21] Michael Haertel, Mark D Hummel, Geoffrey S Strongin, Andrew W Lueck, and Mitchell Alsup. Virtualizing an iommu, November 3 2009. US Patent 7,613,898.

[22] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[23] Saul St John. Thunderbolt: Exposure and mitigation, 2013.

[24] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–34, 2017.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[26] Gil Kupfer, Dan Tsafrir, and Nadav Amit. *IOMMU-resistant DMA attacks*. PhD thesis, Computer Science Department, Technion, 2018.

[27] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[28] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yue-Ting Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[29] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 1. ACM, 2017.

[30] Theo Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon Moore, and Robert Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. 2019.

[31] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–315, 2018.

[32] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[33] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, 2000.

[34] Matt Miller. Trends, challenge, and shifts in software vulnerability mitigation landscape, Feb 2019.

[35] Peter Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, 2005.

[36] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[37] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[38] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. Utilizing the IOMMU scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 549–562, 2015.

[39] Matthew J Renzelmann and Michael M Swift. Decaf: Moving device drivers to a modern language. In *USENIX Annual Technical Conference*, 2009.

[40] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[41] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[42] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[43] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[44] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[45] Gregory T Sullivan, André DeHon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The dover inherently secure processor. In *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, pages 1–5. IEEE, 2017.

[46] Michael M Swift, Brian N Bershad, and Henry M Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1):77–110, 2005.

[47] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[48] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. *IEEE transactions on dependable and secure computing*, 7(1):65–79, 2008.

[49] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B Schneider. Device driver safety through a reference validation mechanism. In *OSDI*, pages 241–254, 2008.

[50] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 818–834. IEEE, 2019.

[51] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, 2006.