A Zero Kernel Operating System: Rethinking Microkernel Design by Leveraging Tagged Architectures and Memory-Safe Languages

by

Justin Restivo

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

February 2020

Author. . . . .
Department of Electrical Engineering and Computer Science
January 29, 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. Samuel Jero
Technical Staff, MIT Lincoln Laboratory
Thesis Supervisor

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

A Zero Kernel Operating System: Rethinking Microkernel Design by Leveraging Tagged Architectures and Memory-Safe Languages

by

Justin Restivo

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 2020, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

A secure kernel is the keystone upon which all software systems are built. Historically, memory corruption errors have accounted for a large portion of kernel bugs. These bugs are difficult to detect and avoid in memory-unsafe languages such as C. To mitigate such bugs, we build on top of an operating system written in a memory-safe language, Rust.

Rust provides memory-safety guarantees while remaining as fast and flexible as other systems languages. Yet, some operations within operating systems, such as hand-written assembly for interrupt handling, do not fit within the scope of a language memory-safety model. To reduce the scope of these errors, microkernels isolate and reduce privilege by moving much of the traditional kernel into userspace services. However, their effectiveness is limited by the inflexibility of modern hardware.

The Zero Kernel Operating System (ZKOS) emphasizes the high-level ideas of compartmentalization and least privileges on a *tagged architecture*. In particular, instead of relying on the Ring model and paging, which coarsely limit privilege and isolation granularity, a tagged architecture allows ZKOS to isolate at the memory word level and provide truly disjoint privileges. To this end, ZKOS slices kernelspace and userspace into fine-grained components based on function. Then, ZKOS defines specific entry and exit points between components and composes policies to limit component transitions and privileges. This increases the precision of isolation and privilege, and complements the local compile-time and runtime checks Rust performs to reduce the scope of bugs.

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. Samuel Jero
Title: Technical Staff, MIT Lincoln Laboratory

Acknowledgments

There are a lot of people to appreciate for aiding me throughout this process. I would like to extend thanks to Howard Shrobe for sparking my interest in systems as my recitation instructor from my days as a sophomore in 6.033 and continuing to mentor me through this MEng years later, and Hamed Okhravi for guiding me through this amazing opportunity and providing support every step of the way. Being a part of this research group that you and Howie have cultivated has made a major impact on my life that I greatly appreciate.

A special thanks Samuel Jero; without his support and limitless patience, this thesis would not have been written. Thanks to Nathan Burow for his insights and vision, Bryan Ward for his mentorship and positivity, Alexandra Clifford for her enthusiasm and willingness to jump in to help, and Richard Skowyra for his thoughtful advice.

Furthermore, I'd like to thank the rest of my research group. Special mentions to Maddie Dawson, Jakob Weisblat, and Jiahao Li for their guidance as I gained my footing at the start of my project, and Jason Priest, who I worked closely with. More include Jennifer Switzer and Claire Nord for sharing their expert presenting advice, in addition to Baltazar Ortiz, Josh Hilke, Elijah Rivera, Alex Huang, Ashley Kim, Rene Garcia, and the rest of my amazing research group (again) for their help and guidance.

A shout out to my friends: without you I wouldn't have made it through MIT or this MEng. Thank you Anne Hunter for your invaluable advice and insight over the past five years. Finally, thanks to my family for the unconditional support they've provided over the course of my life. Especially to my parents, I wouldn't be here without your constant belief in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A secure operating system is a fundamental feature of any computer system. The kernel typically provides strong abstractions around low-level interactions with hardware peripherals and the processor. It must manage system resources and provide a secure Application Binary Interface (ABI) for any software system to interact with hardware. Without a secure operating system, a malicious or buggy application may gain control over these key tasks and thus over the entire computer system. Operating system security is a key focal point of research and the topic of this thesis.

The majority of production-grade operating systems, such as *BSD, Linux, MacOs, or Windows are implemented in C. C is a systems language that provides very low-level features like inline assembly that are required for operating system functions like interrupt handling. Unfortunately, C is a legacy language and memory-unsafe. This opens C to a set of spatial memory errors [1, 2]. Additionally, C's manual memory management can result in an assortment of temporal memory errors. Finally, the various C standards are underspecified and allow for programmers to write functional code that relies on undefined behavior (UB).

Much research has been aimed at detecting or at least reducing the severity of memory errors in kernel code [2]. For example, fuzzing is a popular technique employed to discover such bugs. Fuzzing automatically explores code and data paths in programs by randomly mutating test cases. However, while fuzzing [3–5] aims to provide maximal code coverage, it provides no guarantee of catching most or all errors. Formally verified kernels like SeL4 [6] use mathematical proofs to show the absence of classes of common bugs. However, these proofs are difficult and time consuming

to produce, and make this technique not scalable. SeL4, for example, took thirty person-years to verify around 9,000 lines of code [6]. Other approaches simply seek to mitigate the exploitability of memory errors. A prominent example of this is obfuscating control flow by randomizing the memory layout. Implementations include kernel address space layout randomization (KASLR) in Linux [7] and kernel address space randomized link (KALR) in OpenBSD [8]. However, these techniques remain vulnerable to brute force and information leakage attacks [9–11].

Instead of using the aforementioned techniques to discover or mitigate bugs, memory-safe languages are a promising approach that prevents memory errors with the programming language. Memory-safe languages provide static analysis at compile-time and dynamic checks at runtime to provide memory-safety guarantees, and thus prevent memory errors within the programming language's model. However, a language for kernel implementation must be fast and as flexible as C. Rust [12] combines these qualities. Within regions of code marked as *safe*, Rust provides language-level memory-safety with semi-automatic memory management based on a compile-time borrow-checker [12]. Behavior outside of Rust's programming language model is marked *unsafe* and lacks memory-safety guarantees. Rust maintains equivalence both in performance and flexibility to C [13] via zero-cost abstractions and these unsafe regions, respectively.

Unfortunately, a plethora of bugs such as logic bugs and compiler errors cannot be prevented by memory-safe languages. Additionally, Rust cannot provide safety guarantees for code regions that do not fit within a language-based memory-safety model. In particular, any kernel must contain actions such as inline assembly for handling interrupts or dereferencing arbitrary pointers to interact with memory-mapped I/O (MMIO). These actions do not have meaning in a memory-safety model, and as a result, should not be permitted in memory-safe code. As a result, it is impossible to eliminate unsafe code, and the possibility of memory-safety errors, from a Rust kernel.

Orthogonal to memory-safety, modern architectures provide two main hardware mechanisms to provide separation of privilege: the Ring model and paging. The Ring model forces the operating system to make a hierarchical distinction between kernel Ring code that has complete control over the processor, memory, and peripherals, and userspace Ring code that has far less control. There is very little middle ground for code than needs a subset of the privilege that the Kernel Ring provides. Similarly, paging and the abstraction of virtual memory enforces a coarse-grained isolation model for address spaces as virtual memory provides isolation at the page level.

Least privileges is a foundational security concept commonly applied to kernels by moving all

code that does not require the most privileged Ring into lesser privileged Ring services to reduce the amount of trusted code. Such designs, referred to as microkernels, apply paging to provide isolation. However, these privileges are coarse grained and thus limited due to the hierarchical nature of the Ring model.

A promising solution to these limitations is the use of tagged architectures. A tagged architecture adds several bytes of metadata to each word in memory that encodes semantic information while hardware enforces a flexible set of policies over this semantic information. These policies can include disjoint privilege schemes and fine-grained isolation. They would, for example, allow for some components of the system to have access to MMIO regions of memory, at the word granularity, while completely disallowing access to other components.

1.1 Our Contributions

In this thesis, we propose a novel kernel design, *ZKOS*, that eliminates the majority of memory errors by using a memory-safe language and leverages a tagged architecture to provide fine-grained isolation and a non-heirarchical privilege scheme at low performance overhead.

ZKOS bears some similarity to a memory-safe implementation of a microkernel. However, the trusted codebase of a microkernel may still perform any instruction in Ring 0. Instead, a central idea of ZKOS is to rethink the idea of the Ring model to enforce least privileges in a non-hierarchical manner. We no longer use Rings, but instead partition the kernel into modular *components*, each with a custom set of privileges. We couple these components with a tagged architecture to enforce these custom privileges, resulting in a reduction of privileges for each component. For example, we allow only the trap handling component to execute the `mret` instruction to return to userspace. Then, we model our system with a Finite State Automata (FSM) and use tags to enforce a well-defined control flow between components. Rather than using paging and virtual memory, we use a tagged architecture to extend the granularity of isolation from the page level to the word level.

The remainder of this thesis provides more background, then discusses our novel kernel design, provides an implementation and evaluation of these ideas, and finally considers future and related work.

Chapter 2

Background

This section discusses three main categories of background information surrounding kernel design. First, we examine the benefits and challenges of implementing an operating system kernel in a memory-safe language. Next, we consider hardware isolation features and the types of isolation they can provide. Finally, we review common kernel design patterns and their motivation.

2.1 Memory-Safe Languages for Kernels

Most production-grade kernels are written in C interleaved with raw assembly. C is typically chosen because it contains key low-level features needed for interacting with peripherals and the processor itself. Inline assembly is necessary for managing processor intrinsic assembly instructions that are not part of programming language models. On the x86-64 architecture, for example, reading the time stamp counter, performing an atomic exchange, reading the CPU ID, pushing or popping a stack frame, and configuring interrupt handlers or paging all require specific assembly instructions. Additionally, the creation and use of arbitrary pointers is required for writing peripheral drivers that interact with MMIO [14]. These features are a large part of why operating systems are typically implemented in C.

Because C is low-level, without automated memory management or memory-safety projections, it is vulnerable to dangling pointer dereferences, i.e., temporal errors, and out-of-bounds pointer dereferences, i.e., spatial errors [2]. These memory errors are in practice are quite common in

production kernels and other systems software; for example, Microsoft stated that over 70% of all of their reported bugs were memory errors [15]. The resulting exploits have historically led to fully privileged arbitrary code execution [1, 9–11, 16, 17].

2.1.1 Finding and Fixing Memory Errors

The strongest memory error prevention guarantees come from formal verification of a kernel codebase, as has been done on SeL4. SeL4 demonstrates that formally verifying code in a low-level language such as C is possible, but also incredibly complex. Verifying SeL4 took 30 person years and required many layers of abstractions and at least 100 invariants to verify even a small region of code [6]. With this level of complexity, it remains difficult and time-consuming to formally verify a large codebase.

A plethora of other tools are used to find and fix memory errors in existing C codebases. Fuzzing, the automated generation of test cases through random mutation to maximize code coverage, is the most common method of finding errors. Mainstream static analysis tools such as RATS [18], Yasca [19], or Coverity [20] inspect binaries to detect many common memory errors [21]. However, these tools return a large set of false positives, making their use tedious. Sanitizers such as ASan [22] or Valgrind [23] are another vulnerability mitigating technique that adds inlined reference monitor instrumentation to a binary at or after compile-time [24]. However, depending on where this instrumentation is included, sanitizers cannot link against closed-source libraries or have a large performance cost. Another technique, Symbolic Execution [25], iteratively determines if each path in a program is possible using an SMT solver. This involves solving an NP-Complete problem and blows up exponentially upon encountering common control flow statements such as `for` loops that result in a large number of branch points. This makes symbolic execution computationally infeasible on larger programs [25]. Concolic Execution provides a speed improvement to symbolic execution by executing normally while additionally maintaining symbolic constraints, and then negating these constraints to mutate its input into more test cases [26]. Concolic execution, however, does not find all possible paths, but may be effectively applied to operating systems [26].

Finally, there are several common techniques that focus on increasing the difficulty of exploiting memory errors. Typically, this is done by obfuscating control flow, data and randomizing memory layouts to complicate exploits. For example, randomizing the kernel address space [7] and the code locations [8] are both common ways to hide data. Additionally, research has gone into code

obfuscation methods at both the assembly/binary [27] and programming-language level [28].

2.1.2   Memory-Safe Programming Languages

Desirable qualities for a kernel programming language include C-like low-level capabilities and performance, memory-safety, and performant automated memory management. However, it is difficult for a single language to provide all these features. Low-level capabilities do not fit well with a memory-safety model. Automated memory management such as garbage collection may take a potentially unbounded amount of time and in practice tends to be slow [29]. However, three relatively new languages manage to provide most of these features.

Dlang, Golang, and Rust are all promising choices, each having a mature operating system implementation [30–33]. They remain comparable to C/C++ in speed (less than a two times slow-down on a standard benchmark suite [13, 34, 35]). Golang is a C-like language that implements a concurrent mark-and-sweep garbage-collection algorithm and is memory-safe. For low-level operations, Golang can call raw assembly code. The drawback of the current Golang implementations is its requirement for garbage collection (which can lead to heap exhaustion if the garbage collector needs to allocate memory while freeing chunks) and forced assembly sections [30,36]. Both Rust and Dlang mark sections of code as safe and provide memory-safety guarantees in these sections. They, however, maintain the flexibility of C by allowing unsafe regions of code that lose memory-safety guarantees but can provide functionality such as inline assembly. Dlang provides manual memory management, optional garbage collection (also marked by section), and C-like syntax.

Rust is a unique language that combines high-level aspects of functional languages with the lower-level aspects of C. For example, it allows for C-style inline assembly, macros, and pointers, but also higher-level features like generics, polymorphism, pattern matching, and Haskell-like monadic error-handling with little runtime overhead via zero-cost abstractions. Rust allows semi-manual memory management which is as performant as C. It implements an affine type system [37], which allows for tracking of all structs/objects. This introduces an ownership model over memory which defines an owner and associated scope over each piece of data. When each struct goes out of scope, Rust can determine at compile time to drop the object and then call either the default or a user-defined drop function to free the memory. Rust is built on top of LLVM, so it retains all the optimizations and modern qualities of LLVM and its Intermediate Representation (IR).

15

2.2 Isolation Mechanisms

Traditional architectures rely on paging and the Ring model to provide isolation and privilege to userspace and optionally parts of kernelspace. Tagged architectures can provide more flexible isolation and privilege.

2.2.1 The Ring model

(a) An illustration of the concentric Ring model on x86.

(b) An illustration of paging; contiguous virtual memory address space mapping to physical address space.
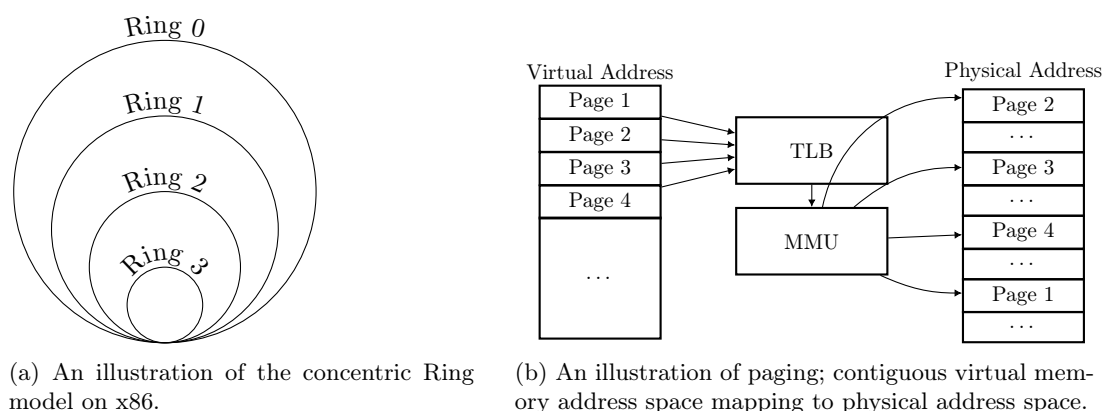
Figure 2-1: Illustration of isolation and privilege hardware enforcement on traditional architectures

The Ring (also denoted mode) model allows code to execute at separate privilege levels. As illustrated in Figure 2-1 (a), these privilege levels are hierarchical and typically thought of as a series of concentric circles (or Rings) in terms of the privileges they contain. For example, in x86 (32-bit) the $(n-1)$th Ring may do everything the nth Ring can and slightly more. Ring 3 has the least amount of privilege; for example, it may not switch Rings, configure paging, register trap handlers, or perform I/O. Intel intended to use Rings 1 and 2 for for drivers. However, while these Rings may access privileged pages, they are not allowed to execute privileged processor instructions. Ring 0 is fully permissioned. The hierarchy described by the Ring model allows for separation of privilege in a coarse, predefined fashion, as the kernel and userspace must be fit into strictly increasingly privileged Rings. Disjoint, rather than hierarchical, Rings would more closely align with the privilege model of a kernel and userspace.

2.2.2 Paging

Virtual memory relies on paging, and provides an entire virtual address space to a process as illustrated in Figure 2-1 (b). This provides two advantages. First, a separate address space provides isolation at the page level. Secondly, paging (implementing virtual memory) allows for non-contiguous regions of physical memory to appear contiguous to the process, thereby increasing memory layout flexibility, and lazy allocation, thereby allowing for less initialization overhead. If a page is accessed by a process but is not currently mapped, the process traps, and the kernel walks the page table to map a physical page into the process's memory.

While paging provides convenient abstractions both for isolation and allocation, these abstractions come at a cost. Each virtual address must be translated into a physical address by walking multiple levels of page tables. While the translation lookaside buffer (TLB) caches the mapping from virtual to physical address, it is often flushed. Additionally, adding a page to this cache is slow as it requires two Ring switches, pushing and popping process registers, and walking the kernel's page tables.

Another aspect of modern paging is the existence of bits to indicate the permissions associated with each page. Specifically, bits allow for the page to be specified as the Cartesian product of read, write, and execute. This allows for data type protection at the hardware level [38]. However, these privileges apply universally on the Ring level. That is to say, a page in an address space is not isolated between all components running in the same Ring until the entire address space is swapped out, which is slow and coarse-grained.

2.2.3 Tagged Architectures

Fundamentally, a tagged architecture adds a set of additional metadata bytes to each word in memory. These bytes can be used to indicate specific information about that word. The architecture further enables the definition of rules about this information. We denote a set of rules a policy, and note that policies may be used to provide fine-grained isolation and a disjoint privilege scheme.

Recent tagged architectures include CHERI [39], TIMBER-V [40], HDFI [41], and Dover [42] among others [43–45]. CHERI's tagged architecture is used as a pointer protection mechanism and complements rather than replaces paging. TIMBER-V adds a 2-bit set of metadata to each word in memory and complements an MPU (*memory protection unit* which provides coarse grained

isolation) to provide a more fine-grained isolation within the Ring model. Each of these architectures complements and tries to improve upon traditional privilege and isolation mechanisms.

The Dover tagged architecture [42, 46] uses a hardware tag on each word of memory, each register, and each control status register (CSR) to encode the address of the actual metadata tag information. This layer of indirection allows for a nearly unlimited number of metadata tags to be associated with each word in memory. These metadata tags may be matched by rules in the Dover policy language to determine whether the current instruction is allowed to execute.

```
1   FirstGrp (mem == [-TagA], code == {TagA} -> env = env )
2   ^ SecondGrp ( mem == [+TagA,+TagB], code == _, csr == _ -> mem = mem, res = {})
3   ^ ThirdGrp  ( code == _, mem == _, env == {TagC} -> fail "execute violation")
```

Figure 2-2: Example of the Dover micro-policy syntax

The Dover tagged architecture is based upon RISC-V. It introduces a secondary core, denoted the Policy EXecution co-processor (PEX), that ensures that each instruction executed by the main core is consistent with the enforced policy. To speed up policy lookup times, Dover includes a policy rule cache next to the processor in a *processor interlocks for policy enforcement* (PIPE) unit.

Additionally, Dover allows for a policy to consist of one or more *micro policies*. Each micro policy has its own set of unique tags and rules. For an instruction to execute, it must be consistant with all rules in the composed policy. This composition allows for intricate and detailed multiplexing of semantic information.

Each provided micro policy executed by the PEX sequentially tries to match the tags surrounding each instruction to a rule in the micro policy, in much the same fashion as a switch statement, except without the fall-through behavior. If a match is found, the PEX applies any actions specified in the rule, including modifying tags or halting the processor. If a micro policy rule match is not found, the micro policy implicitly fails. More specifically, the Dover tagged architecture allows for instruction-level pattern matching upon and modification of the combination of tags observed on the program counter (denoted *env*), the current instruction type, the instruction operands, the tag on memory access, and the currently used CSR, if they exist for that specific instruction.

Next, consider the example micro policy in Figure 2-2. When this micro policy is run, the PEX first tries to match the currently executing instruction with the micro policy rule on line 1. FirstGrp specifies a subset of RISC-V instructions specially tagged with common qualities, such

as referring to memory or writing to a CSR. The syntax states that a match is found if the tags on memory do *not* include TagA, the tags on the current instruction are exactly TagA, and then sets the env tag to itself. If line 1 is not a match, the PEX continues to line 2, which matches on instructions of type SecondGrp with memory including the tags TagA and TagB, and the tags on the current instruction and modified CSRs as wildcards. If line 2 is not a match, the PEX proceeds to line 3, which matches on all instructions of type ThirdGrp with any tags on the instruction and memory, but exactly TagA and TagB on env; if this is a match, the micro policy explicitly fails, halts execution, and provides the "execute violation" error message. If line 3 is also not a match, the micro policy implicitly fails.

2.3   Operating System Design

There are many differing operating system designs, but it is indisputable that *some* privileged code must handle drivers that interface with hardware attached to the processor, such as General Purpose I/O (GPIO) pins, Universal Asynchronous Receiver/Transmitter (UART), etc. Additionally, loading and scheduling userspace applications, performing inter-process communication (IPC), handling traps, providing isolation with virtual memory and paging, providing a system call interface between userspace and kernelspace, and implementing file systems and networking configuration all require some degree of privilege. The division of these tasks involves a complex set of tradeoffs, and there are three major classes of design centered around this division: monolithic kernels tend to be the most popular (examples include Linux and *BSD), microkernels (the prominent example is Apple's XNU kernel), and exokernels.

2.3.1   Monolithic Kernels

As illustrated in Figure 2-3, monolithic kernels group operating system functionality into a single shared address space running in Ring 0, execute userspace in Ring 3, and provide a rich syscall interface between userspace and kernelspace. Because the kernel has a large amount of responsibility, the kernel codebase typically contains large amounts of code which is privileged and thus must be trusted. For example, modern versions of Linux have over 20 million lines of code [47]. The size of the codebase increases the complexity of the system and therefore the number of subtle errors. Additionally, assigning all of these responsibilities to the trusted core of the kernel makes little

19

Ring 3

Userspace

Ring 0

Kernel: {Drivers, IPC, File System, Scheduler, Trap Handler}

Hardware
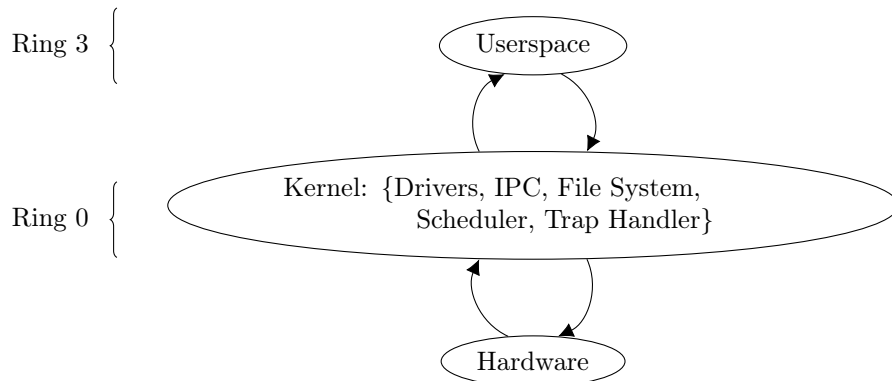
Figure 2-3: Control flow of a monolithic kernel.

sense from a security standpoint; not all these services require Ring 0 or the same address space to perform their task. In fact, most of these services could be isolated from each other and contain only a subset of the privilege provided by Ring 0. Monolithic kernels choose performance over security.

2.3.2 Microkernels

Ring 3

Userspace

File System   Application IPC   Peripheral Drivers

Ring 0

Kernel: {Trap Handler, Scheduler, Service IPC}

Hardware

Figure 2-4: A traditional microkernel design.

Microkernels are an alternative kernel design focused on reducing the amount of privileged kernel code running in Ring 0. As shown in Figure 2-4, microkernels move drivers, application IPC, and

networking out of the kernel into individual services running in parallel with userspace in Ring 3. A microkernel therefore provides a much stronger isolation by giving each service its own virtual address space. Additionally, by merit of running in Ring 3, services are also much more limited in their privileges and more able to deal with service errors, as a driver error will not halt the microkernel.

However, this compartmentalized microkernel design comes with a performance penalty [48]. Services often need to communicate with each other, and each message requires at least two slow context switches into the kernel to change services and address spaces instead of a simple function call as seen in a monolithic design. Each context switch is slow because much overhead is introduced from switching between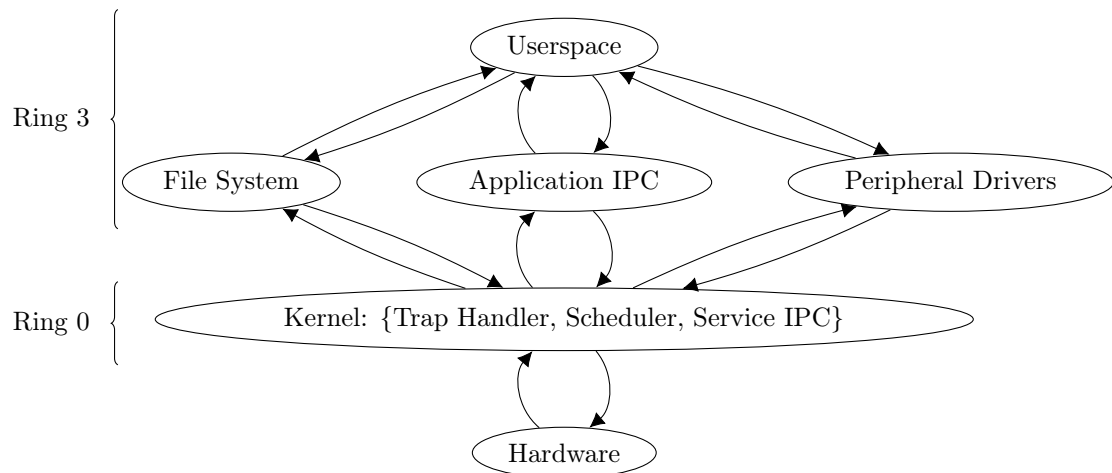 services and includes multiple Ring switches and pushing and popping the register state of both services at least once.

Microkernels isolate most services from each other and restrict their privileges as much as is possible. This is difficult to do on modern hardware, because although services require disjoint privileges, modern hardware enforces privileges hierarchically. Therefore, microkernels design around the hardware to overly restrict privileges to its services in a unprivileged Ring 3. To provide privilege, microkernels require context switches into Ring 0, where a small trusted kernel codebase is able to perform the necessary privileged actions upon request.

2.3.3 Exokernels

Exokernels represent a different approach to component separation where privileged operations are provided as libraries compiled in with each user application. As shown in Figure 2-5, these libraries interact with hardware through *Secure Bindings* running in Ring 0 which expose as direct an API as possible from these bindings to untrusted application libraries running in Ring 3. How these system resources are exposed between applications and their untrusted libraries is determined on first use and thus during runtime there is no privilege verification overhead when resources are accessed by applications. As an example of how this works, physical pages are assigned an owner application upon allocation via a secure binding. At this time, the page may also be marked as shared with other userspace applications. Then, the first time this physical page is accessed by other applications, a secure binding is established between these other applications and the physical memory page in the exokernel, typically with the use of a software implemented TLB. After this secure binding is established, the page is then directly accessible to all userspace applications that have established
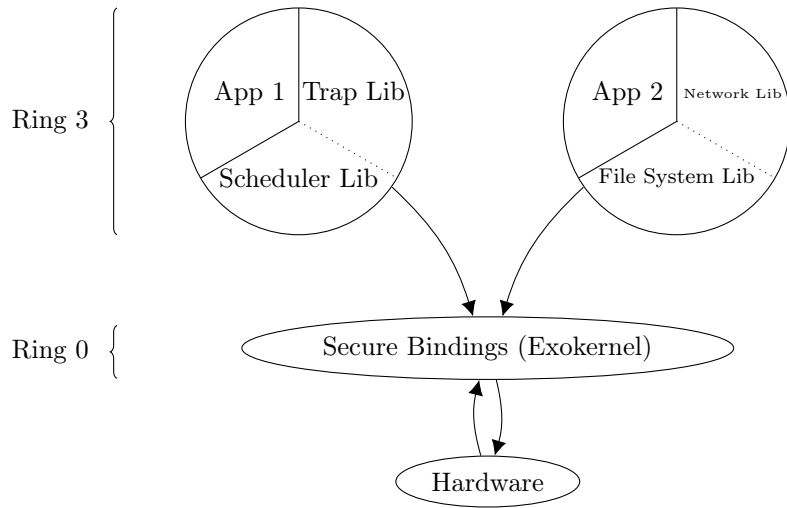
Figure 2-5: Control flow of exokernel

this secure binding with no verification overhead.

While this approach logically separates allocation and use of resources, it places significant trust in userspace applications to manage and share resources appropriately. Suppose a malicious userspace program is given control of scheduling. It may then easily starve other processes. Allocating such high-level privileges to an untrusted application is insecure, but fast, as the only job the exokernel must perform is an initial verification that a library may access a resource. Additionally, exokernels provide abstractions over hardware at the userspace level, which makes portability difficult as every time a program has different peripherals, the abstractions may be different and its libraries must be recompiled for that specific target.

Chapter 3

Design

In this chapter, we provide an overview of our design and discuss our threat model. Then we dive into specific tag micro-policy primitives and their security implications.

3.1 Overview

The main idea behind the design of ZKOS is to more effectively apply least privileges to kernelspace and userspace via a tagged architecture. ZKOS attains this goal by applying policies run on a tagged architecture to small pieces of kernelspace and userspace called *components*. Instead of relying on paging and virtual memory for memory isolation, we implement memory isolation as a policy that provides both significantly finer-grained isolation and the ability to isolate parts of kernelspace from each other, similar to how a microkernel would. Instead of using Rings to provide separation of privilege, ZKOS uses a policy to enforce a strict privilege scheme upon each component. This allows for ZKOS to apply a fine-grained, disjoint privilege model to both userspace and kernelspace. Through the use of a tagged architecture and policies, ZKOS improves security and performance by more fully realizing least privileges.

ZKOS is able to provide fine-grained, word-level memory isolation in kernelspace and userspace due to a tagged architecture's ability to tag and apply policies at the word level instead of the page level. Policies can specify exactly which pieces of memory the current component should and should not be able to access. This decreases memory fragmentation while increasing flexibility

with how much memory is allocated to a component. Additionally, components no longer need to live in separate address spaces as in microkernels. Instead, ZKOS can provide the same isolation guarantees as paging but within a single address space.

This design has important performance implications. Despite being generally accepted as a necessary technique, paging has a measurable performance impact. The Singularity project [49] estimated that a traditional microkernel incurs overhead of near 33% compared to the same code executing in a single address space in Ring 0. Since ZKOS provides memory isolation without virtual memory while remaining in a single address space, ZKOS eliminates the need for paging. Additionally, this allows for ZKOS to enforce memory isolation in kernelspace. Unlike traditional microkernels that can simply modify their page table entries to access arbitrary words, ZKOS manually prevents the kernel from accessing other component's state.

ZKOS also leverages a tagged architecture to provide disjoint privileges. Recall that the Ring model is hierarchical; lower privilege modes must include all actions of higher privilege modes. In contrast, ZKOS splits control over privileged actions into many discrete, disjoint privilege sets. These privileges can then be parceled out independently to different components that need them, without regard for hierarchy or ordering. For example, ZKOS is able to confine to a initialization component the ability to initialize the trap handler address. We implement these privileges via a policy on a tagged architecture that imposes restrictions upon instruction execution. This is incredibly powerful on a large scale such as a kernel, as it allows for the specification of exactly the permissions each component needs. Note that this in not inherently incompatible with the Ring model; ZKOS simply uses the tagged architecture to manage privileges while running all code in Ring 0.

An interesting side effect of not depending on Rings for userspace/kernel space separation is that system calls now become function calls to specific entry points into the kernel. This should incur noticeably less overhead than a full syscall, since it removes the slow and expensive operations involved in context switching. As a result, ZKOS replaces syscalls with simple function calls for transitioning between components, a convention we refer to as a *sysfunc*.

Using these novel isolation primitives, ZKOS splits a traditional kernel's tasks into specific components. Then, tags are applied to each component to provide isolation and separation of privilege. Each word of a component is tagged in such a way that it may only be accessed or executed by that specific component. ZKOS then defines specific transition points between components to

replace syscalls. This technique is illustrated in Figure 3-1.
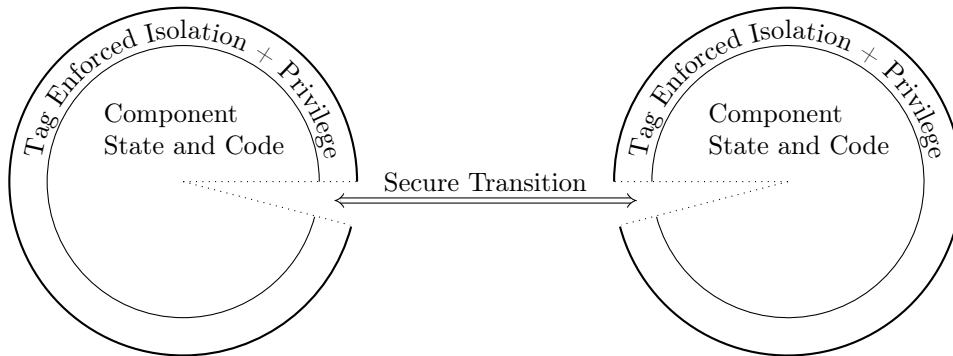
Figure 3-1: Conceptual overview of ZKOS implementation

Throughout ZKOS's design, we sought to follow two key design principles. The first is security; we aim to design a system that provides effective isolation and limitation of privilege to thereby reduce the effective impact of bugs. Second, we aim to make our system at least as performant as traditional designs such as a monolithic kernels on simple userspace benchmarks.

The remainder of this chapter first introduces component isolation, privilege management, and transition primitives using micro policies on a tagged architecture, and then applies these policies to design ZKOS.

3.2   Threat Model

We assume that since Rust code is not formally verified, any Rust codebase will contain logic and compiler bugs. Second, we assume that the kernel codebase is not inherently malicious, but may contain logic bugs. We further note that userspace applications are untrusted and potentially malicious. Finally, we assume that our tagged hardware is implemented correctly and does not contain bugs. We are therefore able to trust tags and policies.

3.3   Memory Isolation and Privilege Separation

ZKOS provides fine-grained memory isolation, thereby allowing each component to access an specific number of variably-sized chunks of memory. Furthermore, ZKOS provides a granular set of

privileges to each component; each component may only execute instructions that are required for execution.

We isolate a component via tags instead of via classic techniques such as paging or segmentation. This approach contrasts favorably with these existing memory isolation techniques. Segmentation limits an executing component to a few segments of any size in memory. This is severely limited due to the finite number of segments and the consecutive nature of a segment (defined by an offset and bounds). Paging marginally improves upon segmentation by providing many non-consecutive regions of memory to a process. Unfortunately, paging administers only coarse grained isolation, with a lower bound of 4KB pages on x86. ZKOS's tag based memory isolation combines the desirable features of both these forms of isolation: the variable size of segments and the extensibility and non-consecutive nature of paging. Furthermore, ZKOS avoids the downsides of each method. ZKOS is not subject to the consecutive memory and a small number of segments constraints of segmentation or the fixed size constraints of paging. ZKOS can also duplicate with tags paging's ability to mark data as read, write, and execute, which ZKOS can also duplicate with tags.

Traditional paging does provide one abstraction not implemented in ZKOS: a separate address space per component. We choose not to implement this feature for simplicity's sake; however, if desired, a mechanism such as paging could be introduced into ZKOS to complement tags with the associated abstraction of a large virtual address space. We reason that since Position Independent Code (PIC) or Position Independent Executable (PIE) allows us to run applications and use shared libraries within the same address space without recompilation, and since ZKOS provides isolation through a tagged architecture, a separate address space per component is unnecessary from both an isolation and usability perspective. Instead, ZKOS runs within a single, flat address space.

The remainder of this section discusses the specifics surrounding the techniques ZKOS uses to implement memory isolation and privileges. ZKOS uses micro policies to compartmentalize the kernel and isolate each component and to enforce specific privilege limitations upon components at a more detailed level than paging's read, write, or execute bits or the Ring model.

3.3.1   Isolation Policy

The goal of this micro policy is to isolate each component of userspace and kernelspace in a more granular manner than paging. ZKOS defines exactly what data and code exist within each component at the word level. ZKOS tags this data and code as belonging to a component at compile

RAM	TAG
STACK	
DATA	Kernel
TEXT	
...	...
STACK	
DATA	App 2
TEXT	
...	...
STACK	
DATA	App 1
TEXT	

Figure 3-2: High level view of component tag initialization for isolation

time, and then applies these tags to memory when the kernel binary is loaded. After tagging, the RAM resembles Figure 3-2. To these initial tags, ZKOS then applies a policy during kernel execution that, for each instruction executed within the component, requires all data reads and writes to have the component's tag. The tags on the env register (also called the PC tags) are used to determine the component in which ZKOS is currently executing. This design provides an isolation mechanism more granular than paging, because an arbitrary number of both contiguous and non-contiguous words may be tagged. ZKOS precisely identifies exactly what data and code belongs to a component at compile-time, and isolates as desired.

A second consideration of component isolation is sharing and transferring data between components. This is similar to mapping a page into two address spaces. Since ZKOS does not currently share data, neither is implemented. However, we present two possibilities for future investigation. The first method would be to, at compile-time, mark words as owned by both components. Alternatively, a second method would be to, at run-time, include a micro policy that allows for system calls into a component with the ability to share between components by executing instructions that add the relevant component's tag to each word. While the second approach is far more flexible, the first approach is faster and more scalable.

27

RAM	TAG
RODATA	READ
STACK	READ,WRITE
DATA	READ, WRITE
TEXT	EXECUTE

Figure 3-3: High level view of component tag initialization for RWX privileges

3.3.2 RWX Privileges Policy

Next, we consider a micro policy primitive for privilege. We start with a micro-policy primitive that duplicates and expands upon the data-type privileges enforced by paging. It provides memory privileges within a given component by providing specific tags for read, write, and execute permissions and allowing these permissions to be combined in any order on a memory region. As a result, all memory regions will have a granular purpose. This provides more flexibility than most paging schemes, by providing the full Cartesian product of permission options. During loading of the ZKOS binary, each word is tagged with the desired permissions for that code or data. Figure 3-3 illustrates permissions over a typical component. We tag the read only data section of a ZKOS component with only the READ tag, both the stack and data sections with READ and WRITE as the component will need to modify both, and the text (code) section with EXECUTE as the text section may only be executed.

Now that we have explained how tag initialization works, we consider the micro-policies enforced at runtime. ZKOS enforces the following rules based on tag: load instructions requires a READ tag, any write instructions require a WRITE tag, and all executing instructions require the EXECUTE tag. Thus, we check that a load instruction's operand contains a READ tag, a write instruction operand contains a WRITE tag, and that all instructions that are executed are tagged with an EXECUTE tag. If any of these conditions are violated, the policy fails and execution is halted. These rules finely demonstrate the enforcement of specific actions to each data type just as paging does.

While ZKOS implements solely RWX privileges, it is possible to extend this paradigm to a more detailed data view of the exact number of reads, writes, or executes a component is allowed to perform upon a word in memory. This could be useful if, for example, ZKOS were to implement a database that wished to implements write once read many (WORM) at the hardware level.

3.3.3   Privilege Policy

ZKOS enforces further privilege limitations with policies implemented by a tagged architecture. To this end, ZKOS adds specific privileges to each component through extensions to the aforementioned isolation and RWX privileges policies. ZKOS defines privilege over a broader set of resources: use of MMIO, CSRs, and special registers and instructions. This is a far richer definition of privilege than typically provided by the Ring model and paging, which does not usually include register or MMIO restrictions and rigidly defines the limitations of CSRs based on Ring on RISC-V.

To enforce privilege limitations upon MMIO, ZKOS initializes MMIO regions with tags at the word-level of READ and WRITE in the RWX policy. CSRs and special registers are analogously individually tagged as readable or writable. During execution, additional rules are added to check for the expected tags on the CSRs, special registers, and MMIO in the RWX policy. After this policy is configured, it manages read and write privileges for a component far more precisely than paging.

However, ZKOS must enforce these privileges for all components, instead of a single component. To do so, we multiplex this privilege policy with the isolation policy. This allows for the isolation policy to determine if the component is allowed to perform an action or access a region, and the privilege policy to enforce, on a per-component basis, the specifics of the privileges over memory, CSRs, and MMIO.

Execution of special instructions are privileged actions and therefore are enforced by the isolation policy over a specific region. More specifically, additional elements in the isolation policy allow for specific instructions to be executed *only* if they are run in a specific component. This is implemented by adding more policy rules that match on special instructions and only succeed if env contains a component tag that is allowed to execute a special instruction.

Together, these two primitives provide a much stronger definition of privilege than that implemented by the Ring model and paging. Not only are components and their privileges completely disjoint instead of the hierarchical privileges enforced by the Ring model, but also they provide a much richer and more flexible definition of privilege than the Ring model including word-level privileges over MMIO and individual CSRs, and component-level privileges over instructions. Since ZKOS may have an arbitrarily large number of components instead of the usual four Rings (as on x86), this is a large improvement in the granularity of privilege separation by hardware-enforced security.

3.4 Sysfuncs

RAM	TAG
Component B Entry Point: `_enter_b:   ...`	B, AB_transition
...	...
Component B Text	B
...	...
Component A Text	A
Component A Exit Point: `j _enter_b`	A, AB_transition

(a) Tag Representation of Sysfunc
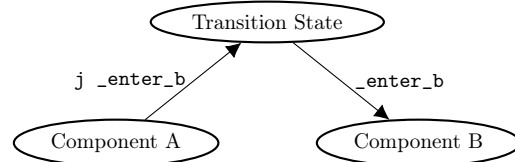
(b) State Machine Representation of the Transition

Figure 3-4: Transitioning from component A to component B with a sysfunc

In this section, we discuss the policy specifics for transitions between components. At a high level, we would like these transitions to be equivalent to switching isolation contexts and privileges (similar to both swapping address spaces with virtual memory and switching Rings, respectively). Additionally, for some component transitions such as syscalls, we must communicate data describing the action the calling component expects the recipient component to perform. ZKOS's solution is a class of specially marked function calls, called *sysfuncs*. The name denotes their similarity to syscalls, with the addition of the much smaller overhead of function calls. We apply this concept to transition between arbitrary components in ZKOS.

ZKOS implements these sysfunc transitions with tags. Suppose during execution, ZKOS needs to transfer from component A to component B. ZKOS transitions by marking the exit point of the first component (a jump instruction) and the entry point of the second component with the same unique tag at compile-time. Then, ZKOS includes a policy that states that when executing inside component A if a jump instruction containing this unique tag is encountered, to move to a special transition state identified by a unique tag on the PC. The next executed instruction must also contain this unique tag, which then completes the transition between components, including switching isolation contexts and privilege levels, and allows execution to resume under the second component. This is illustrated both in RAM and with a Finite State Machine view of the component transition in Figure 3-4.

This allows for well-defined transitions between components, in almost the same way as a context

switch, except without incurring the overhead of switching Rings, pushing and popping the full register state (only a few registers are necessary), and flushing the TLB to switch address spaces.

This explicit marking of exit and entry points to components is both flexible and secure. It allows for not only a bijection between entry and exit points but also a surjection. This allow for a detailed description of transitions between components within a policy. These transitions are implemented in ZKOS as augmentations to the isolation policy.

3.5   ZKOS Components

Now that we have discussed the isolation, privilege, and transition primitives used by ZKOS, we can detail its overall design. This entails discussion of each component and its function and the transitions between components.

As shown in Figure 3-5, ZKOS has five components with disjoint privileges. The *userspace* component runs userspace applications and has no privileges. The *trap* component has only the required privileges for determining a trap's cause and marking the trap as handled. The *handler* component has no privileges and handles userspace syscalls. The *kernel* component consists of all drivers and the scheduler. As a result, this kernel component has the requisite privileges for interacting with peripheral hardware and scheduling processes. Finally, ZKOS contains a *bootstrap* component with the necessary privileges for driver, process, and trap initialization. The boundaries were drawn for these components because each performs a distinct, fundamental task and requires a specific, disjoint set of privileges. This allows ZKOS to showcase, as a proof of concept, the security benefit provided to the kernel by tags. The specific privileges and isolation of each component are implementation specific and thus discussed in the implementation section.

We now consider the high-level transition points between components that should exist in ZKOS. First, we consider transitions between ZKOS and a userspace application; the equivalent of a syscall. In the case of a yield system call, the control flow should return to the kernel and be redirected to another process. In all other cases, ZKOS determines the type of syscall in the handler component and then handles the syscall with the relevant driver. Finally, ZKOS jump back into the userspace component.

Next, consider traps. We have removed the syscall exception, but other traps may still occur such as a timer interrupt or a misaligned instruction exception. For these traps, we enter the trap
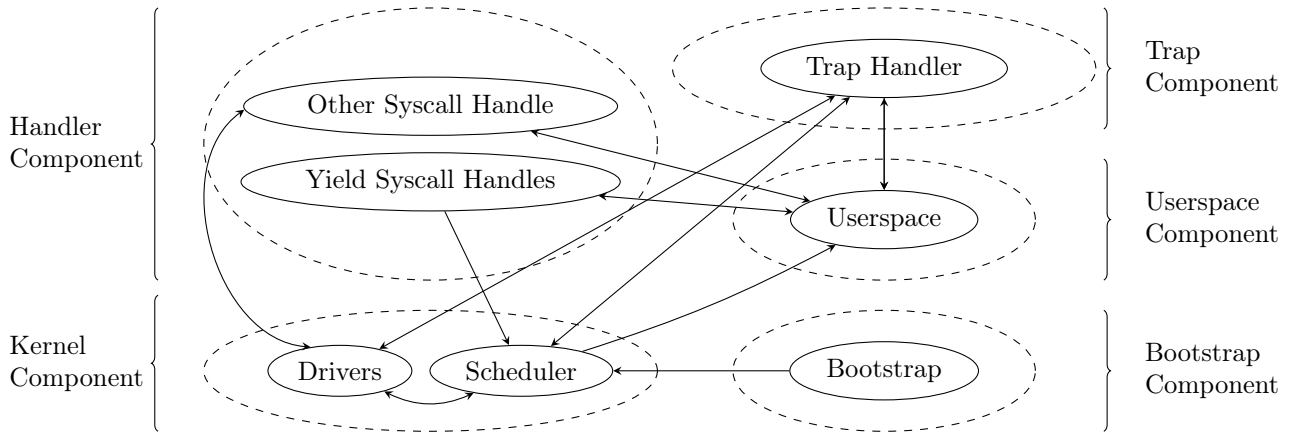
Figure 3-5: Complete ZKOS system diagram

handler from any point in execution (either from an exception or a userspace interrupt). The trap reason is determined within the trap handling component, then kernel code is called to handle the trap. In the case of a timer interrupt, ZKOS must transfer control back to the kernel component in the form of the scheduler. ZKOS only enables unmasked interrupts in the userspace component, so it knows to restore userspace tags to that component upon return. In the case of exceptions, ZKOS remains in the trap component and panics. Currently, ZKOS only supports a single application, but isolating between userspace applications may be added by using a policy that changes the component based on tags on return address.

Consider the remaining components. Execution starts with the highly permissioned bootstrap component, which performs initialization. The bootstrap must transfer control to the kernel component, which then schedules processes and resumes process execution. The kernel component contains both drivers and the scheduler.

Chapter 4

Implementation

We begin with a discussion of Tock OS [31], the starting point of ZKOS. Next, we discuss how tagging works for ZKOS. Then, we discuss implementation of ZKOS's micro policy primitives and conclude with a discussion on implementing the ZKOS kernel.

4.1   Tock OS

Tock [31] is an operating system implemented in Rust for ARM (and now RISC-V) aimed at embedded IoT systems. Because Tock is written in Rust, it provides strong safety guarantees over a large portion of its kernel codebase.

Part of the effort behind this thesis was porting Tock to RISC-V. Specific contributions involved an implementation of the CLINT (core level interruptor, a RISC-V interrupt-handling peripheral), debugging and partial rewrites of existing RISC-V context switching assembly, a CSR abstraction library, work on a PMP (physical memory protection unit–another RISC-V peripheral akin to ARM's MPU) driver, debugging of userland crt0 to be able to run a C userland on RISC-V, and debugging of UART, GPIO pins, and RTC (real time clock) peripheral drivers.

Conceptually, Tock is somewhat a hybrid between the monolithic and microkernel designs. Userspace applications run in the architecture's equivalent of Ring 3, and as a result are prevented by hardware from performing privileged actions. The entire OS is one monolithic binary running in Ring 0, like monolithic kernels. However, userspace applications trap into *capsules* as the first

level of system call handling. Tock forces all capsules to consist entirely of safe Rust by including a *safe* keyword at the start of each capsule's file. Capsules then both handle syscalls and implement high-level abstractions over drivers. By virtue of safe Rust, capsules provide strong static safety guarantees at the language level that have significant similarity with the isolation guarantees found in microkernels. These capsules then either call other high-level capsules also written in safe Rust, or low-level drivers written in unsafe Rust that perform the necessary unsafe actions to configure and interact with peripherals and other hardware. Tock's goal is to minimize the amount of low-level driver code and thus the net amount of unsafe code.

The design of Tock makes it a good starting point for ZKOS. A kernel implemented in mostly safe Rust removes common memory errors from the majority of the kernel. Additional design choices such as Tock's minimization of the unsafe code regions, modular abstractions that may be extended to components, and relatively small codebase make it a suitable starting point for ZKOS.

The remainder of this section first covers implementation details about how ZKOS uses tags. Next, we cover implementing the ZKOS micro policy primitives. Finally, we conclude with a discussion of the specifics of ZKOS's components privilege on RISC-V.

4.2 Tagging

As ZKOS is highly dependent on a tagged architecture, one of the most critical decisions was the tagged architecture upon which to implement. ZKOS requires a flexible and powerful language in which to describe policies, and requires the use a fairly large number of tags. The Dover tagged architecture fulfills both these requirements and is therefore the architecture upon which we chose to implement ZKOS.

Dover currently provides two ways to initialize tags on words in memory: by ELF section and by address range. To initialize tagging, ZKOS includes a python script that iterates over and tags each ELF section of the compiled ZKOS binary. Unfortunately, ZKOS relocates certain sections of the binary, which means our script needs to rely on tagging the address ranges to which these sections will be relocated. This works well for isolating components.

ZKOS also requires marking transition points between components. Unfortunately, neither tagging by section or by address range works well for tagging these transition points. To mark transition points, we manually searched the ELF binary for, and tagged, the function call addresses. This was

not scalable, as it requires non-negligible effort to change the addresses upon each recompilation. A better solution would be to have Rustc and LLVM encode address tags into ZKOS binary, which we leave to future work.

Additionally, we need to tag CSRs and MMIO. The Dover tagged architecture allows for CSRs to be individually tagged by address. Similarly, MMIO tags may be initialized by address range. This form of initialization worked well in ZKOS.

4.3   Implementing ZKOS Primitives

By leveraging the Dover Policy language, we were able to develop tag micro policies that successfully implement the three policy primitives required for ZKOS. We now discuss specific examples of how ZKOS implements its isolation, RWX privileges, and transition primitives. We do not discuss further privilege implementation, as the granular privileges required by ZKOS are natural extensions of the isolation and RWX privileges micro policies with little conceptual implementation difference.

```
1  custom_silo_pol =
2   loadOrStoreGrp(mem == [-User1], env == [+User1] -> fail "user 1 memory violation")
3   ^ loadOrStoreGrp(mem == [-User2], env == [+User2] -> fail "user 2 memory violation")
4   ^ loadOrStoreGrp(mem == [-Kernel], env == [+Kernel] -> fail "kernel memory violation")
5   ^ allGrp(code == [+User1], env == [+User1] -> env = env)
6   ^ allGrp(code == [+User2], env == [+User2] -> env = env)
7   ^ allGrp(code == [+Kernel], env == [+Kernel] -> env = env)
8   ^ allGrp(code == _, env == _ -> fail "not supported!" )
```

Figure 4-1: Kernel and userspace isolation policy with two user applications

First, we discuss how ZKOS implements an isolation micro policy. Figure 4-1 showcases a simplified isolation policy between two userspace application components and a kernel component. Lines 2 to 4 guarantee that accessing memory that does not share the same component tag as the current component will cause an exception. Lines 5 to 7 state that if the currently executing instruction's tag and memory accesses match the component's tag, then ZKOS will explicitly allow the instruction. Finally, if the env tag or memory and code tags are in an inconsistent state, an exception will occur with the catch-all on line 8.

Next, we show an example micro policy to implement RWX privileges in Figure 4-2. Lines 2 to 4 explicitly fail if load locations lack the READ tag, write locations lack the WRITE tag, or

35

```
1  custom_rwx_pol =
2    loadGrp(mem == [-READ] -> fail "read violation")
3    ^ storeGrp( mem == [-WRITE] -> fail "write violation")
4    ^ allGrp ( code == [-EXECUTE] -> fail "execute violation")
5    ^ loadGrp  (code == [+EXECUTE], env == _, mem == [+READ] -> res = {}, env = env)
6    ^ storeGrp (code == [+EXECUTE], env == _, mem == [+WRITE] -> mem = mem, env = env)
7    ^ allGrp(code == [+EXECUTE], env == _ -> env = env)
```

Figure 4-2: Read, write, execute privilege micro-policy

any instructions lack the EXECUTE tag. Lines 5 to 7 explicitly succeed if loads have a READ tag, writes have an WRITE tag, and instructions have an EXECUTE tag. If ZKOS fails, it is able to provide a specific error message describing the reason for failure. Remember that we would multiplex this policy with the isolation micro policy to provide isolation and RWX privileges on a per-component basis.

Finally, we provide an example policy to illustrate how component transitions work in Figure 4-3. We define three distinct tags for components A, B, and transitioning between A and B. Then we pattern match: on lines 2 and 3, ZKOS executes normally for instructions that are tagged with A or B. On line 4, if ZKOS is in component A and the instruction is a jump containing the transition tag, ZKOS switches to a transition state. On line 5, ZKOS lands on an instruction that is both in component B's code region and contains the corresponding transition tag indicating that this is a valid entry point to component B to complete the transition from component A to component B. Line 6 is an explicit failure indicating that ZKOS did not jump from a valid entry point to component B. To include multiple transitions, we augment the isolation policy with this primitive for each unique component transition.

```
1  transition_policy =
2    allGrp(env == {A}, code == {A} -> env = {A})
3    ^ allGrp(env == {B}, code == {B} -> env = {B})
4    ^ jumpGrp(env == {A}, code == {A,AB_transition} -> env = {AB_transition})
5    ^ allGrp(env == {AB_transition}, code == [+B,+AB_transition] -> env = {B})
6    ^ allGrp(env == {AB_transition}, code == _ -> fail "illegal entry point")
```

Figure 4-3: Sysfunc micro-policy

4.4 ZKOS Implementation and Privileges

To create ZKOS, we refactored Tock to fit into the distinct components according to our design. The main refactoring was centered around trap handling; the trap and syscall handles were split out of the scheduler's control flow. Additionally, the assembly in the handler component required for switches between the separately compiled userspace and handler components was included as a Rust macro library used by the handler component. Modifications were also included in userspace to support function calls instead of the `ecall` instruction. The kernel and bootstrap components were already logically separate in Tock.

In the remainder of this section, we consider each component separately, determine the specifics of their isolation and privilege, and then consider transition points between components. In terms of isolation, we determine what memory each component may read, write, and execute. By privilege, we discuss which CSRs and MMIO may be written to and read from, and which components may write to the GP (global pointer) register. Tables 4.2, 4.3, 4.4, and 4.1 present an overview.

First, we consider the userspace component. The code for the userspace component is contained within the text section of a separately compiled ELF binary. As ZKOS does not have a file system, this ELF is obj-copied along with the other sections of the userspace ELF and location metadata generated by Tock into the kernel ELF binary, and then executed by ZKOS at runtime. We know where the userland text sections exists beforehand and can tag it at compile-time with our eython script. Similar reasoning applies to the stack and grant regions[1]. This is performed at compile time, as we lack the ability to dynamically load tags. The tags on these userspace components follow the tag policies discussed in the design section. In terms of privileges, userspace should not be able to read or write any CSRs, MMIO regions, execute special instructions, or modify the GP.

Next, we consider the trap component. The trap handling code resides in its own ELF section that we tag and is provided its own stack. The ZKOS trap handling component calls into the trap handlers while remaining within the trap component, or panics in an infinite loop, depending on whether the trap type is a interrupt or an exception. The implemented trap handlers include a UART and a timer. We isolate the trap component to be able to only read from and write to its own stack and execute its own code region.

The trap component has the privilege of reading and writing to `mstatus` and `mie` to disable

---

[1] The grant region is a Tock alternative to the heap shared between kernelspace and userspace.

interrupts so that it can handle the currently called interrupt. This component also needs to read `mcause` to identify the cause of the trap, and to read and write to `mscratch` since it stores the address of saved application state there. Finally, this component needs to read and write to `mepc`, which contains the address of the trap. This allows ZKOS to recover from the trap. For MMIO, the trap component must mark that it has handled interrupts; as a result, it requires read and write access to the CLINT and PLIC MMIO regions. Finally, the trap handler must be able to return after dealing with the trap and so must be allowed to execute the RISC-V specific `mret` instruction.

Next, we consider the handler component. The key purpose of this component is to define a syscall ABI for ZKOS. We do this by hardcoding the address of the entry point of each ABI call into the userspace linker file. Our isolation policy gives the handler component its own stack and code sections within ZKOS's ELF. However, the handler component must call kernel code to handle traps, so we provide it the ability to execute the kernel component's code region. Permission-wise, similar to the userspace component, the handler component does not need any special permissions.

The kernel component contains all capsules, drivers, and the scheduler. Its code is contained within the text region of the ZKOS ELF; its data lives on the kernel stack. Because the kernel component contains drivers for both the timer and the UART, it requires read/write privileges over those regions of MMIO. The kernel component must handle suppressed interrupts, which means that it must also have read/write access to the CLINT and PLIC MMIO.

We required a bootstrap component that performs a large amount of initialization of peripherals and processor specific functionality as ZKOS boots. This component has significant privilege because it must initialize the UART and timer drivers, set the initial state of `mstatus`, initialize the GP register for optimization purposes, disable interrupts in kernelspace with `mstatus` and `mie`, and configure the address of the trap handler with `mtvec`.

Because the bootstrap component needs to make calls across the whole kernel, it is only weakly isolated from the kernel. In particular, the bootstrap component needs to make calls to much kernel component code, so we allowed the bootstrap component to execute both the kernel component's text section and its own start text section and kernel entry section. Additionally, we allowed the bootstrap component to share the kernel component's stack and the userspace grant region as it initializes much state used by the kernel and userspace programs.

We now turn to consider the necessary transitions between components. Userspace components must be able to transition to either the trap handler component in the case of an interrupt, or the handler component to handle system calls. In the former case, ZKOS is configured such that unsuppressed interrupts may only be generated in userspace and ZKOS can therefore return to userspace directly from the trap handler component. In the latter case, the transition points for the five syscalls implemented by Tock are explicitly identified in the operating system with tags.

The kernel and handler components must be able to transition into the userspace component. These specific transition points are marked and tagged, as described in the design section. The bootstrap component is only allowed to transition to the kernel component at a single location, and is not allowed to return to the bootstrap component as we no longer need to perform any initialization.

Component	MMIO								Special Instructions		Registers GP		
	UART		Timer		PLIC		CLINT						
	r	w	r	w	r	w	r	w	ecall	mret	r	w	x
User	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Trap	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓	✗	✗	✗
Kernel	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗
Handler	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Bootstrap	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✗

Table 4.1: MMIO, instruction, and register permissions

Component	CSR													
	mstatus		mcause		mscratch		mtvec		mtval		mie		mepc	
	r	w	r	w	r	w	r	w	r	w	r	w	r	w
User	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Trap	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✓	✓	✓
Kernel	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Handler	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
BootStrap	✓	✓	✗	✗	✗	✗	✓	✓	✗	✗	✓	✓	✗	✗

Table 4.2: CSR permissions

Component	ELF Section														
	Kernel Text			Kernel Stack			User Text			User Grant			User Stack		
	r	w	x	r	w	x	r	w	x	r	w	x	r	w	x
User	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✗	✓	✓	✗
Trap	✗	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Kernel	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Handler	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗
BootStrap	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗

Table 4.3: ELF section permissions

Component	ELF Section														
	Sysfunc Stack			Sysfunc Text			Start Text			Kernel Entry			Trap Text		
	r	w	x	r	w	x	r	w	x	r	w	x	r	w	x
Userspace	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Trap	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
Kernel	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗
Handler	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
BootStrap	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✗

Table 4.4: ELF section permissions

Chapter 5

Evaluation

We evaluate our design based on a qualitative analysis of its security and a quantitative analysis of its performance when compared to the Tock operating system.

5.1 Security

We provide a qualitative approach to analyzing ZKOS's security. In particular, we discuss how ZKOS equals or exceeds the security of traditional kernels in three key ways. Its implementation language reduces memory errors, its use of tags improves upon both virtual memory by providing a more granular isolation and the hierarchical Ring model by providing disjoint privileges. We showcase examples of each improvement.

Our implementation language choice protects against the majority of memory errors. In the Linux kernel, the language provides no guarantees about the safety of any of its over 20 million lines of code. The result of this highly complex code without any language protections is a very high number of memory-safety bugs (over 10,000 buffer overflows alone [50]). By choosing Rust and Tock as our base, memory errors can only occur within unsafe regions. The lines of unsafe code in ZKOS compose just 6.65% of the codebase[1]. This prevents memory errors within the majority of ZKOS. Spatial and temporal errors like buffer overflows may only occur within these memory-unsafe regions; this drastically reduces the scope of memory bugs.

---

[1]Based on numbers generated by cargo-count: https://github.com/kbknapp/cargo-count

Our compartmentalization policy provides memory isolation similar to that which virtual memory offers. Suppose we have an arbitrary physical address, `0xdeadbeef`. With virtual memory, a service may not access `0xdeadbeef` by default. Similarly with tags, if `0xdeadbeef` is not tagged as owned by the component, tags prevent access. In this respect, tags provide the same isolation guarantees as virtual memory. However, suppose we wish to provide `0xdeadbeef` to a processes. With virtual memory we would have to map an entire page including `0xdeadbeef` into the virtual address space. With tags, ZKOS can simply tag the word to indicate that the executing component may access it. We implemented this statically at compile time in ZKOS by marking each component with a corresponding tag. Therefore, in terms of isolation, ZKOS and traditional virtual memory both provide the same level of isolation.

Virtual memory also provides read, write, and execute permissions on memory. Our RWX policy makes the same distinction, except at the word level of granularity and with greater flexibility. In other words, if we wished to mark `0xdeadbeef` as executable and writable, we could provide both tags and our policy would provide this functionality. Therefore tags provide a stronger level of granularity–both at the word and permissions level for *each component*. This allows us the flexibility to implement, for example, write⊕execute as a security policy if desired both in ZKOS and userspace.

We demonstrated that our design is equivalent to virtual memory in terms of isolation and enforcing memory type by testing the policies in ZKOS. To test component isolation, we wrote code in each component that attempted to access words in memory that did not belong to that component. To test the RWX policy, we attempted to use memory in a way it was not intended (such as a read on execute-only memory) in each component. Execution stopped in these scenarios.

ZKOS also improves upon the hierarchical privilege design typically used in monolithic and microkernels. In particular, ZKOS splits the kernel into separate, isolated components and provides each component with only the privilege it requires. With only four core kernel components, we are able to demonstrate the enhanced security benefit. We have a stronger privilege scheme and have made privilege escalation attacks more difficult. An adversary might be able to exploit logic errors in ZKOS, but since we have split the privilege between components, escalation of privilege is much more difficult.

5.2 Performance

In this section, we perform an analysis of the performance of ZKOS. We compare with Tock both because it was the basis for our work and because of its traditional monolithic design.

Unfortunately, we do not yet have access to a hardware implementation of the Dover tagged architecture. We do have an emulator (based on QEMU), but it is not cycle accurate for policy decisions. This severely limits our ability to measure meaningful performance results for the whole system.

We run simple userspace programs to benchmark the performance of ZKOS and Tock on an untagged 32-bit RISC-V FPGA. Additionally, we micro-benchmark ZKOS's sysfuncs and Tock's syscalls to observe exactly the performance benefit ZKOS provides. Next, we benchmark the policy rule cache performance of ZKOS on the tagged architecture emulator to determine how well ZKOS utilizes a policy cache. These two evaluations give us some idea of the performance of ZKOS.

5.2.1 Sysfunc Benchmarks

One of the key changes in ZKOS was replacing syscalls with sysfuncs and eliminating Ring switching overhead. In this section, we analyze the resulting performance improvements. We evaluate this in two ways: benchmarks between ZKOS and Tock over simple userspace programs, and micro-benchmarks over single syscalls. This allows us to estimate the incurred overhead of a Ring switch in both a typical use case and in isolation.

To ensure that our benchmarks are as accurate as possible, we compile both Tock and ZKOS with the same version of the Rust compiler [2], and all userspace programs with the same version of the gcc toolchain [3]. For an accurate comparison, we base ZKOS and its userspace changes on top of the same version of Tock [4] and Tock's userspace [5] that we test against. To make our results more accurate, we remove Tock's preemptive scheduling and suppress all interrupts in userspace. We run all of our tests on an Arty A7 [6], which runs the SiFive E21 core [7].

We construct several examples of typical userland applications to benchmark. The first bench-

---

[2] Rustc version: nightly-2019-09-23

[3] GCC Toolchain with RISC-V support, version 8.3.0

[4] Hash c29f893a3c3f2bb7a2a41755fbd23346440a8d7f of https://github.com/tock/tock

[5] Hash 4c250e1f7722d2663b6ba0e9eb4fdba46e2050e8 of https://github.com/tock/libtock-c

[6] Purchased here: https://store.digilentinc.com/arty-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/

[7] Described here: https://www.sifive.com/cores/e21

mark program, C_HELLO, prints "hello world" via UART. This benchmark exercises all of our sysfuncs and Tock's syscalls. Our second benchmark, 5_EMPTY_SYSCALL, repeatedly calls an empty sysfunc or syscall. The purpose of this benchmark is to more accurately measure the overhead of a Ring switch over many sysfuncs and syscalls. Our third benchmark, BLINK, toggles GPIO pins once. Similarly to the UART example, this benchmark reserves memory and exercises all sysfuncs/syscalls of the LED driver. Finally, we include a fourth benchmark, NO_SYSCALL, that simply runs the userspace crt0 (which initializes the userspace stack, heap, global offset table (GOT), and data sections and involves several syscalls), so we can estimate the overhead that a crt0 induces.

We run each benchmark 10,000 times, with a 10 iteration burn-in and observed no more than a two cycle standard deviation from the average of the runs. Additionally, we can provide a theoretical bound on our sample error using the following theorem[8], independent of the underlying distribution of the data:

Theorem 1 *Let $A_1 \cdots A_N$ be IID random variables ranging in $[0, 1]$, and let $p \triangleq \mathbb{E}[A_i]$. Then, for every $\epsilon \in (0, 1]$, it holds that:*

$$\mathbb{P}\left(\left|\left(\frac{1}{N}\sum_{i=0}^{N}A_i\right) - p\right| < \epsilon\right) < 2 \cdot \exp\left(\frac{-\epsilon^2 N}{4}\right)$$

In our case, we assume each run is a sample of the ith random variable in the set of N IID variable $X_i \in [0, 0x1000000]$[9]. We normalize X_i then apply Theorem 1 with ϵ of 200 cycles and N of 10000 samples to get the result[10] that our estimator (the average of all 10,000 samples) is within 200 cycles of the expected value of X with probability 0.995.

Figure 5-1 (a) illustrates the average number of cycles taken by each benchmark when run in both ZKOS and Tock. Figure 5-1 (b) illustrates how the overhead of syscalls scales when scaling BLINK to toggle multiple GPIO pins. In each case, ZKOS is at least 10% faster. Something as small as a CRT0 is 10% faster with sysfuncs, while 100 GPIO pin toggles is about 20% faster. These results strongly suggest that using sysfuncs noticeably improves performance in ZKOS.

---

[8]A generalization of Chernoff bounds proved as Corollary 9 here: http://www.wisdom.weizmann.ac.il/~oded/PDF/pt-apdx.pdf

[9]our benchmarks are short and therefore fewer than 0x1000000 cycles

[10]the error calculation: https://www.wolframalpha.com/input/?i=2*e%5E%28-200%5E2*10000%2F%284*0x1000000%29%29

(a) Average cycle estimator for userspace benchmarks between ZKOS and Tock

(b) Average cycle estimator for GPIO benchmarks between ZKOS and Tock

Figure 5-1: Userspace program benchmarks

(a) Timing of four context switches

(b) Timing of two context switches
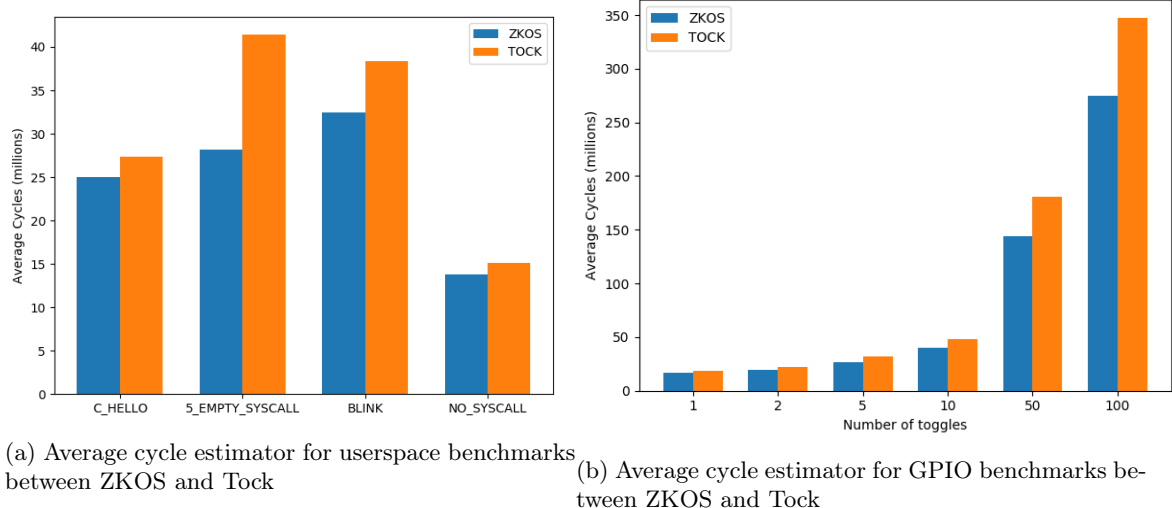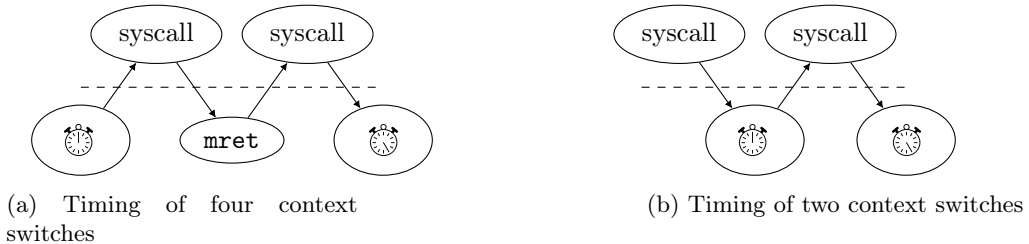
Figure 5-2: The two types of micro-benchmarks between ZKOS and Tock

We also benchmark the overhead of a single context switch to more precisely observe the performance impact of sysfuncs. Unfortunately, our board does not implement either a userspace cycle counter or userspace timer. As a result, we were limited to recording cycles exclusively in the kernel. To work around this limitation, we employ two approaches, involving multiple context switches. In the first series of benchmarks, four consecutive context switches are timed, and in the second series of benchmarks, two consecutive context switches are timed. The control flow of the two micro-benchmarks are shown in Figure 5-2. In part (a), a syscall starts the timer, an empty syscall is executed, and then the timer is toggled via another syscall to determine elapsed time. In part (b), a timer is started, the kernel returns into userspace, and then another syscall immediately toggles the timer. Figure 5-3 suggests that the context switching overhead from kernelspace to userspace and back decreases by around 45% when using sysfuncs. This is large performance increase and

reflects Singularity's 33% performance increase from running userspace within Ring 0 [49].
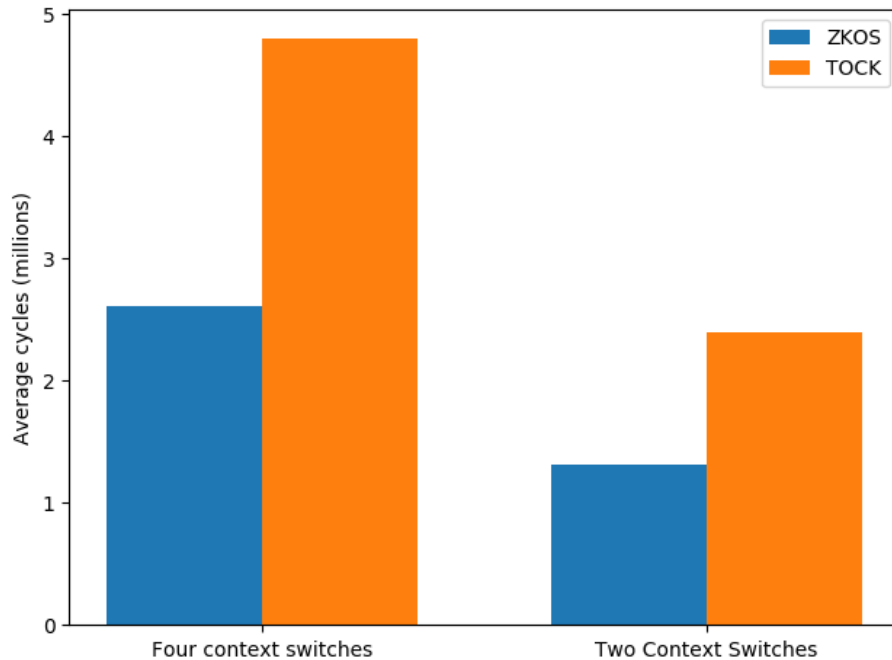

Figure 5-3: Average cycle estimator for micro-benchmarks

5.2.2   Tag Cache Benchmarks

The Dover tagged architecture includes a policy rule cache to decrease lookup times for recently used policy rules. Each time a combination of tags is observed when executing an instruction, a mapping from that sequence of tags to a policy rule is loaded into the policy cache. The next time that same combination of tags is observed, the corresponding action may be drawn directly from the policy cache. Caching is accurately emulated in the Dover fork of QEMU[11] and emulates a least recently used (LRU) eviction policy. While the Dover tagged architecture simulator is not cycle accurate, it does provide accurate information on tag cache miss/hit rates, which represent a large part of the tagged architecture's runtime overhead.

We evaluate the tag cache miss rate of ZKOS with three userspace programs over several cache sizes. The first program prints to UART, the second program toggles a GPIO pin 100 times, and

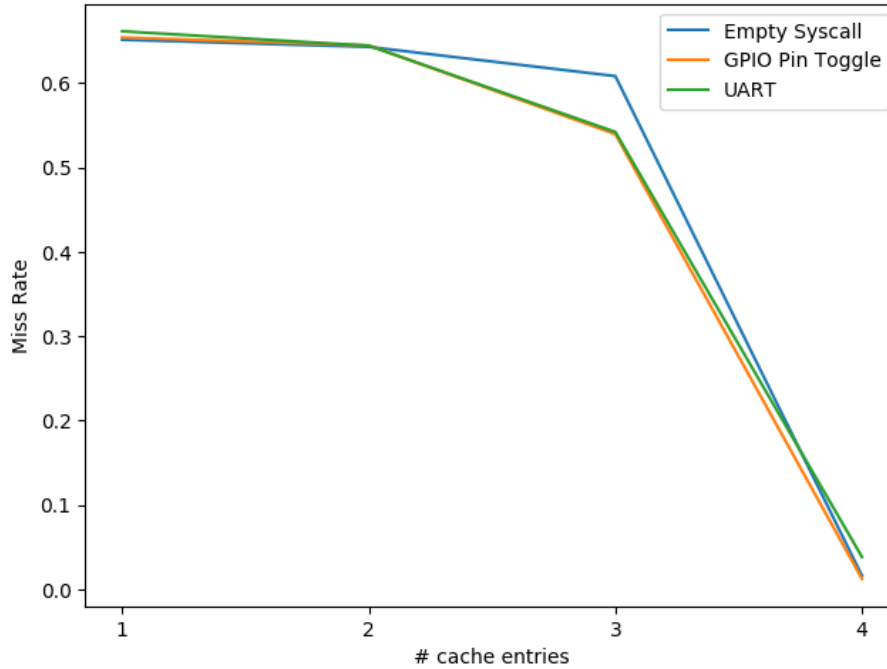[11]https://github.com/draperlaboratory/hope-qemu

46

Figure 5-4: Cache benchmarks

the third program makes 100 empty syscalls. In Figure 5-4, we observe the fractional cache miss rate on these programs as the cache size increases from 1 to 4. We found that upon increasing the size of the rule cache beyond 4 resulted in a cache miss rate of around 0.2%; these misses we attribute to initially loading the policies into the cache.

These results suggest the overhead of ZKOS on the Dover tagged hardware will be small, as ZKOS's miss rate is small. Each component executes thousands of instructions on the same rule before switching components. Therefore, even use of a cache with a number of entries similar to the number of components results in a high hit rate. The empty syscall benchmark with a cache size of 3 illustrates the opposite case. Here, there are 4 components in frequent use, but only 3 cache slots. As a result, the cache hit rate was noticeably lower than other scenarios with either fewer components or fewer transitions. However, once the commonly used rules for all components could both fit into the cache (in this case, a cache size of 4), the miss rate becomes negligible. Additionally, Dover's FPGA implementation should have a policy rule cache with on the order of 1000 entries which is far more than ZKOS's policies require.

Chapter 6

Discussion

ZKOS opens the way for significant future work in four primary directions. First, we could increase the componentization and disjointness of privilege in ZKOS. Second, we could improve evaluation with realistic benchmarks on real hardware. Third, we could increase automation of the tooling needed for ZKOS. Fourth, we could expand on the testing framework for ZKOS.

First, we may further the application of least privilges to ZKOS. The bootstrap component is currently over-privileged. It could be split into multiple, less-privileged components with fewer privileges. A set of drivers could also be separated out of the kernel, each of which may only access the specific portion of MMIO designated to it. This would include the UART, GPIO pins, and the timer included with the CLINT. Additionally, the scheduler could live in its own isolated component and its privilege could be reduced to deciding which process to run next. ZKOS could also share the same stack and state between components; this would be more efficient than having stacks on a per-component basis. Also, the introduction of a policy to manage this data at the stack-frame granularity could further enforce isolation at the function level in addition to the component level. Finally, we would like to isolate data shared between components with tags. For example, we would like the kernel state to be shared between the bootstrap and kernel components to only be accessible from these components. This could be done by allocating and tagging this state at compile time instead of dynamically allocating it on the stack.

Another promising direction for future work would be the evaluation of ZKOS on a real hardware

implementation of the Dover tagged architecture using a standard benchmark suite such as beebs[1] or CoreMark[2]. To do this, we would need to increase the maturity of ZKOS's userland library and sysfuncs significantly, correct a GOT issue with unsupported PIC code within LLVM's assembly generation for RISC-V, add multithreading to ZKOS, integrate multiple userspace application support into ZKOS, and acquire an FPGA or ASIC implementation of the Dover tagged architecture. We would then be able to more directly compare the performance of a typical monolithic kernel such as Linux running on the Dover tagged RISC-V with paging and a TLB to ZKOS on tagged RISC-V without paging.

A third direction would be the addition of Rustc compiler intrinsics to both indicate sysfunc transitions between ZKOS components and to indicate the code and data that are accessible by specific components. This would allow us to have a far simpler implementation, as the compiler would automatically mark methods and transition points during compilation instead of the currently unscalable approach of tagging specific addresses, address ranges, ELF sections, symbols and segments after compilation in a python script. This would also allow us to mark generics after they were monomorphized, which would allow us to finely mark code regions. Currently, Rustc disallows any sort of marking upon a code region that is implemented over generics, making it difficult to scalably mark monomorphized code regions.

A final direction for future work would be to further test the isolation and privileges of our design. We acknowledge that short of formal verification, precisely testing the correctness of both tag initialization and policies is a difficult problem, as it may be subject to subtle logic errors. Inclusion of a fuzzing framework that would insert permission errors into component code regions within ZKOS and expect execution to halt would mitigate this problem instead of the currently unscalable method of manually written patches to ZKOS. Secondly, we would like error handling in ZKOS to be recoverable. Instead of halting execution upon a policy violation, we would like to modify our trap handler to catch and resolve an exception, and then resume execution if possible.

---

[1]http://beebs.eu/
[2]https://www.eembc.org/coremark/

Chapter 7

Related Work

In this thesis, we look to provide better isolation via a tagged architecture and a novel kernel design. This section discusses related work around alternate ways of dealing with memory-safety errors, alternative operating system designs, and other uses for tagged architectures.

Memory errors compromise 70% of Microsoft's security-relevant bugs in the past 20 years [15] and have tens of thousands of reported CVEs across a huge variety of software. A variety of research efforts have been aimed at mitigating these issues. Fuzzing [3–5] generates many test cases to try and trigger memory errors while static analysis tools look for common exploits but generate many false positives. Sanitizers instruments programs with inlined reference counters to ensure instructions are not allowing vulnerabilities [24]. Symbolic [25] and Concolic [26] execution look at all or many paths, respectively, but fail to scale to large programs. Other efforts obfuscate control flow or data [7, 8, 27, 28].

Other operating systems use programming languages to provide static guarantees that improve security. For example, Singularity [49] developed a formally verifiable, memory-safe language based on C# called Sing# and then used it to implement 90% of a kernel. Singularity used language level guarantees to provide isolation instead of hardware mechanisms. In a similar vein, BiscuitOS [30] and CliveOS [36] are implemented in Golang and provides all the benefits of memory-safety. Unfortunately, these kernels also incur significant memory and speed overhead due to garbage collection. Tock [31] and RedoxOS [32] are performant kernels written in Rust with small regions of unsafe code composing under 8% of both kernels; this allows them to be both performant and mostly

memory-safe.

Microkernels historically have designed around hardware to provided security. Mach [51], the first example of moving traditionally kernel responsibilities such as the file system into userspace services, was slow due to IPC between components. L4 microkernels [52, 53] have improved upon Mach on a variety of architectures in both security and performance. SeL4 [6],the most recent L4 microkernel, is aimed at embedded devices, and provides formal verification across its C codebase to improve security. Another class of kernels, exokernels/unikernels, move kernel services into userspace libraries. [54, 55]

Tagged architectures have been used for a variety of security purposes. A prominent example, CHERI [39, 56, 57] introduces a special extended pointer called a *capability* that includes a tag-enforced metadata about its object to complement the privilege of virtual memory and isolation of the Ring model. While this prevents memory errors, CHERI retains the hierarchical privilege of the Ring Model. Baggy Bounds [58] implements a similar idea of object bound checking with pointers for memory error prevention. These Baggy Bounds are implemented with bits typically reserved for paging instead of tags.

Tags have also been used to provide enclaves and secure operating systems. TIMBER-V [40] combines a 2-bit tag on each word in memory with an MPU and Rings to duplicate the isolation found in enclaves at the hardware level with small overhead. However, this application is heavily userspace focused and kernel bugs still remained a problem. TIARA [45] implements a compartmentalized operating system in a Ring-less, single address space system via a supporting tagged architecture. TIARA applies least privileges to each compartment and defines how these compartments should interact. TIARA enforces permissions via function call *gates* that handled as a middle-man to handle interactions between compartments that retains privileges from both compartments. While conceptually similar to ZKOS, TIARA differs from ZKOS by being designed for garbage collection, its tagged architecture, and its specific implementation details.

Chapter 8

Conclusion

Traditional operating systems provide coarse-grained isolation in the form of paging and virtual memory, and hierarchical privileges with the Ring model. These techniques are slow by nature and force operating systems such as microkernels to make design decisions that sacrifice security for performance. ZKOS explores alternative isolation and privilege techniques via the Dover tagged architecture.

ZKOS replaces paging and the Ring privilege model with a fine-grained isolation and disjoint privileges enforced with tags. Much like a microkernel, ZKOS applies least privileges to compartmentalize the kernel into five isolated components, each with disjoint privileges. Then, ZKOS utilizes the Dover tagged architecture to granularly enforce component isolation and disjoint privileges for each component. This helps to confine the impact of logic errors within a single component and makes privilege escalation significantly more challenging. ZKOS uses a sysfunc primitive to transition between components; this primitive avoids the overhead incurred by invalidating the TLB and Ring switches.

ZKOS demonstrates that combining a tagged architecture, least privileges, and a memory-safe language can create a secure operating system. Additionally, our evaluation of ZKOS's performance suggests that ZKOS will perform well on hardware due to both sysfuncs and its high policy rule cache utilization.

Bibliography

[1] Smashing the stack for fun and profit by aleph one. available at https://insecure.org/stf/smashstack.html.

[2] 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.

[3] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[4] 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. ACM, 2017.

[5] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[6] 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. ACM, 2009.

[7] Kaslr linux. available at https://lwn.net/Articles/569635/.

[8] Karl openbsd. available at https://marc.info/?l=openbsd-tech&m=149732026405941&w=2.

[9] CVE-2017-5012. Available from MITRE, CVE-ID CVE-2017-5012., January 2 2017. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5012.

[10] CVE-2019-4154. Available from MITRE, CVE-ID CVE-2019-4154., January 3 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-4154.

[11] CVE-2017-11085. Available from MITRE, CVE-ID CVE-2017-11085., July 7 2017. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-11085.

[12] Rust compiler. https://github.com/rust-lang/rust. Accessed: 2020-01-3.

[13] Rust versus c. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/gcc-rust.html. Accessed: 2019-10-10.

[14] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Combined Volumes 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4: System programming Guide*, 2011.

[15] Microsoft: 70 percent of all security bugs are memory-safety issues. https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/. Accessed: 2019-12-1.

[16] CVE-2019-11893. Available from MITRE, CVE-ID CVE-2019-11893., June 7 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11893.

[17] CVE-2019-14896. Available from MITRE, CVE-ID CVE-2019-14896., August 10 2019. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-14896.

[18] The rough auditing tool for security. https://security.web.cern.ch/security/recommendations/en/codetools/rats.shtml. Accessed: 2020-01-3.

[19] Yet another source code analyzer. https://github.com/scovetta/yasca. Accessed: 2020-01-3.

[20] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. Analysis tool evaluation: Coverity prevent. *Pittsburgh, PA: Carnegie Mellon University*, pages 7–11, 2006.

[21] Rahma Mahmood and Qusay H Mahmoud. Evaluation of static analysis tools for finding vulnerabilities in java and c/c++ source code. *arXiv preprint arXiv:1805.09040*, 2018.

[22] Address sanitizer. https://github.com/google/sanitizers/wiki/AddressSanitizer. Accessed: 2020-01-05.

[23] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[24] 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.

[25] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[26] Su Yong Kim, Sangho Lee, Insu Yun, Wen Xu, Byoungyoung Lee, Youngtae Yun, and Taesoo Kim. Cab-fuzz: Practical concolic testing techniques for {COTS} operating systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 689–701, 2017.

[27] Gregory Wroblewski. General method of program code obfuscation. *N/A*, 2002.

[28] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009.

[29] Tyler Hannan, Chester Holtz, and Jonathan Liao. Comparative analysis of classic garbage-collection algorithms for a lisp-like language. *arXiv preprint arXiv:1505.00017*, 2015.

[30] 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.

[31] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The case for writing a kernel in rust. *Proceedings of the 8th Asia-Pacific Workshop on Systems - APSys17*, 2017. doi:10.1145/3124680.3124717.

[32] Redox os. https://www.redox-os.org/. Accessed: 2019-12-01.

[33] Powernex. https://github.com/PowerNex/PowerNex. Accessed: 2019-12-01.

[34] Go versus c++. https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/go-gpp.html. Accessed: 2019-10-10.

[35] Rust versus c. https://github.com/kostya/benchmarks. Accessed: 2019-12-1.

[36] Francisco J Ballesteros. The clive operating system. Technical report, Universidad Rey Juan Carlos, 04 2014.

[37] Aaron Weiss, Daniel Patterson, Nicholas D Matsakis, and Amal Ahmed. Oxide: The essence of rust. *arXiv preprint arXiv:1903.00982*, 2019.

[38] Hp report on data execution prevention. http://h10032.www1.hp.com/ctg/Manual/c00387685.pdf. Accessed: 2020-01-05.

[39] Robert N.M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical report, University of Cambridge, Department of Computer Science and Technology, 08 2019.

[40] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. Timber-v: Tag-isolated memory bringing fine-grained enclaves to risc-v. In *NDSS*, 2019.

[41] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[42] Gregory T. Sullivan, Andre Dehon, Steven Milburn, Eli Boling, Marco Ciaffi, Jothy Rosenberg, and Andrew Sutherland. The dover inherently secure processor. *2017 IEEE International Symposium on Technologies for Homeland Security (HST)*, 2017.

[43] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 487–502. ACM, 2015.

[44] Udit Dhawan, Nikos Vasilakis, Raphael Rubin, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and André DeHon. Pump: a programmable unit for metadata

processing. In *Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy*, page 8. ACM, 2014.

[45] Howard Shrobe. Tiara: Trust-management, intrusion-tolerance, accountability, and reconstitution architecture. *NICECAP*, Sep 2009.

[46] Arthur Azevedo De Amorim, Maxime Dénes, Nick Giannarakis, Catalin Hritcu, Benjamin C Pierce, Antal Spector-Zabusky, and Andrew Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830. IEEE, 2015.

[47] The linux kernel archives. https://www.kernel.org/. Accessed: 2019-10-10.

[48] Christoph Lameter. Extreme high performance computing or why microkernels suck. In *Linux Symposium*, page 251, 2007.

[49] Galen C. Hunt and James R. Larus. Singularity. *ACM SIGOPS Operating Systems Review*, 2007.

[50] Buffer overflow query cve database. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow. Accessed: 2019-11-17.

[51] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. *NA*, 1986.

[52] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[53] Abi Nourai. A physically-addressed l4 kernel. *BE thesis, University of NSW, Sydney*, 2052, 2005.

[54] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[55] Anil Madhavapeddy and David J Scott. Unikernels: Rise of the virtual library operating system. *Queue*, 11(11):30, 2013.

[56] Brooks Davis, Robert NM Watson, Alexander Richardson, Peter G Neumann, Simon W Moore, John Baldwin, David Chisnall, James Clarke, Nathaniel Wesley Filardo, Khilan Gudka, et al. Cheriabi: Enforcing valid pointer provenance and minimizing pointer privilege in the posix c run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 379–393. ACM, 2019.

[57] Robert N.m. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, and et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. *2015 IEEE Symposium on Security and Privacy*, 2015.

[58] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*, pages 51–66, 2009.

[59] Cătălin Hriţcu. Micro-policies: Formally verified, tag-based security monitors. In *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, pages 1–1. ACM, 2015.

[60] Doaa Abdul-Hakim Shehab and Omar Abdullah Batarfi. Rcr for preventing stack smashing attacks bypass stack canaries. *2017 Computing Conference*, 2017.

[61] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security and Privacy Magazine*, 2004.

[62] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 2009.

[63] Nick Roessler and André DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 478–495. IEEE, 2018.

[64] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[65] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[66] Udit Dhawan and André DeHon. Area-efficient near-associative memories on fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(4):30, 2015.

[67] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.