Securing Operating Systems using Hardware-Enforced Compartmentalization

by

Yianni Giannaris

B.S., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2021

Author . . . .
Department of Electrical Engineering and Computer Science
August 6, 2021

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

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

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

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

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

Securing Operating Systems using Hardware-Enforced Compartmentalization

by

Yianni Giannaris

Submitted to the Department of Electrical Engineering and Computer Science
on August 6, 2021, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

Monolithic kernels have been the traditional design choice of many modern operating systems for practical and historical reasons. Though monolithic systems excel in performance, they suffer from exposure to security vulnerabilities. The past 6 years of published Linux CVE data has revealed hundreds of security vulnerabilites that can potentially be exploited by an attacker to escalate privileges and leak sensitive user data. Though some of these vulnerabilites can be mitigated with proper memory safety enforcement, others require privilege separation to ensure code only accesses data that is explicitly granted by a developer. We present Hardware-Assisted Kernel Compartments (HAKC), a solution that mitigates exposure to security vulnerabilities by leveraging modern commodity Arm hardware and automatic LLVM instrumentation to enforce compartmentalization in an effective manner without requiring significant developer effort. Using Arm Pointer Authentication Codes (PAC) and Arm Memory Tagging Extensions (MTE), HAKC enforces a two-tier compartmentalization scheme that is performant and provides flexibilty for up to $4 * 10^{15}$ compartments, which, when compared to prior works, is orders of magnitude more compartments afforded to developers. To test HAKC, we implemented a compartmentalization policy for nf_tables, a commonly used packet filtering LKM. LKMs are prime targets for compartmentalization because CVE analysis has shown that most kernel vulnerabilites reside in LKMs, and the HAKC two-tiered compartmentalization scheme easily adapts to LKM logical groupings of kernel subsystem functionality. Evaluations show that we are able to acheive strong security enforcement without adding significant overhead.

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

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

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

Acknowledgments

Thank you Savvas, Mom, Dad, Yiayia and Papou. I truly believe that whatever success I may lay claim to would have remained a figment of my imagination without your never-ending love and support.

Uncle Rick and Aunt Ellie, thank you for cheering me on. I doubt I will ever again meet friends who share your kindness, generosity, and perseverance. Uncle Rick, I don't know if you will ever read this, but if you do I think you would agree that it's not too bad for a code monkey.

Derrick, thank you for showing me the ropes, and patiently dealing with my naive questions. I'll be saving that beer to share in person.

Nathan, Hamed, and Howie, you provided me with orders of magnitude more attention and care than I deserved. I cannot express how thankful I am for your support and the opportunities you gave me to grow through the challenges I faced this last year.

Contents

Chapter 1

Introduction

The monolithic kernel architecture has been the de facto design standard in operating systems since the inception of Unix in 1969. By allowing most kernel code to share memory and run in hardware privileged mode, this style of architecture enables functionality, performance in terms of speed, and ease of development [1]. However, these advantages come with the cost of a large bug-prone code base with few mechanisms for component isolation. As a result, any vulnerability in the operating system compromises the entire system.

Operating system vulnerabilities pose major risks in monolithic operating systems because the kernel generally has full control of user space processes. This implies that an attacker with access to kernel space can compromise a user's passwords, cryptographic keys, and other sensitive data. Today, Linux, a monolithic operating system that was originally intended for experimental PC operating system development, is used for billions of applications, ranging from high performance computing, to PC's and IoT devices, including surveillance cameras, routers and modems, smartwatches, android based mobile devices, fitness trackers, smart TV's, among other things [2]. The progression of IoT has resulted in a larger and ever increasing reliance on computing devices connected to the internet. This fact, combined with the ubiquitous use of Linux and other monolithic operating systems is a *serious* cause for concern and motivation for further research in mitigating operating system vulnerabilities.

Software and hardware defenses have been developed to mitigate kernel vulnerabilities. Some software protections such as KAISER [3] and KASAN [4] help enforce memory safety. Various control flow integrity schemes (CFI) have been developed as software protections against ROP attacks (Section 2.1.2). Other defenses such as Intel's SMAP/SMEP [5] utilize hardware to prevent the kernel from accessing and executing user data and code, thereby enforcing privilege separation. These defenses are only capably of thwarting a subset of possible attacks, as some attacks will leverage programmer errors to change program behavior in a manner that is completely within the specification as defined by the programmer (see Section 2.1.3). Some solutions to combat programmer error include formal verification of programs, concolic execution, and use of memory safe languages (Section 2.4,Section 2.5,Section 2.6). Unfortunately, each of these solutions comes with a cost, making there application in modern day kernels challenging.

Another approach to mitigating the impacts of kernel vulnerabilities is to devise mechanisms that enforce *privilege separation*. Privilege separation refers to minimizing code data access privilege to only that which is necessary for correct execution of a program. *Compartmentalization* is the concept of enforcing privilege separation by segmenting a software system into various components with minimized interdependence. A privilege policy, either manually or automatically defined, then controls interactions between *compartments* (we will use the word compartment and partition interchangeably). We can define *isolation* as an extreme form of a compartmentalization scheme in which there is zero communication between compartments. Applications supported by the same operating system that do not share data would be considered isolated. Through compartmentalization, system exposure to vulnerabilities is limited to the component or module containing the vulnerability, thus providing protections against malicious attackers who could not otherwise be thwarted with traditional software defenses.

Linux contains kernel source code findings, such as developer error or security weaknesses, that can be exploited by a malicious attacker (*i.e.,* compromising *user-to-user* and *user-to-kernel* isolation, or unintended user-privilege escalation). To help understand the nature of mitigating such vulnerabilities, the last 6 years of Linux

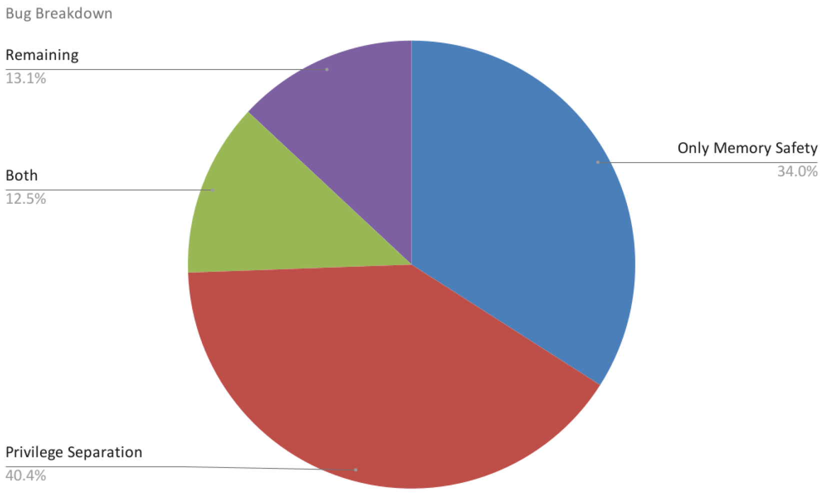


Figure 1-1: 567 Linux kernel vulnerabilities from 2015-May 2021. Note that *Remaining* is comprised of vulnerabilities caused by logic errors, race conditions, and integer overflow

reported CVEs were analyzed and categorized by whether memory safety or privilege separation would be required to mitigate the vulnerability. These decisions were base on textual analysis (*i.e., arbitrary code execution* falls under privilege separation, while *use-after-free* falls under the memory safety).

Figure 1 − 1 describes the CVE analysis findings. The results show that 193 known Linux kernel vulnerabilities can be mitigated *only* with proper memory safety enforcement, and *only* proper privilege separation enforcement can provide defenses against 229 known vulnerabilities. Furthermore, 71 of known vulnerabilities require either privilege separation *or* memory safety. These observations are the primary motivation behind exploring the efficacy of using kernel compartmentalization to enforce privilege separation.

The Linux source code is comprised of a relatively small core of functionality, such as memory mapping and context switching logic, along with a relatively large amount of peripheral functionality, such as networking support, filesystem support, and device drivers. This peripheral functionality can be compiled as a Linux Kernel Module (LKM), and be dynamically linked to the kernel image upon necessary use. Of the 567 vulnerabilities analyzed, 301 were found in the `drivers/` and `sound/`

directories of the kernel source. Since most of the code in these two directories is intended to be used as part of an LKM, we can safely conclude that the majority of vulnerabilities are located in LKMs.

Software-based compartmentalization schemes have been proposed — such as microkernel architectures or isolation through sandboxing — but generally incur non-negligible overhead and/or do not enforce compartmentalization at the granularity required to mitigate fine-grained attacks (such as intra-page control-oriented attacks Section 2.1.2). In contrast, hardware defenses are generally more performant, as the responsibility of enforcing compartmentalization is enforced through architectural functionality. However, until recent developments, conventional hardware mechanisms have lacked support for compartmentalization schemes at the granularity required to thwart sub-page size attacks in a performant manner. For example, mechanisms such as virtual addressing only compartmentalize at the granularity of individual pages. Other mechanisms such as the Ring Privilege Model, incur significant overhead upon switching privilege modes (Section 2.7). Though past research efforts have focused on improving hardware mechanisms to support fine-grained compartmentalization, most tested schemes are implemented in emulation and have not been adopted in commercial hardware.

Arm has recently released two hardware extensions, Arm Pointer Authentication Codes (PAC) and Arm Memory Tagging Extension (MTE) [6, 7]. PAC is an architectural extension that is intended to be used to cryptographically enforce code-pointer integrity (CPI). MTE is an architectural extension that is designed to be used as a lock-and-key mechanism for pointers accessing data.

We present Hardware-Assisted Kernel Compartments (HAKC), a novel compartmentalization framework that uses these two new additions to the Arm architecture to effectively compartmentalize the Linux kernel in a fine-grained manner. HAKC provides developers with a compartmentalization API that allows developers to define a two-tier compartmentalization policy consisting of cliques and compartments. Cliques represent logical designations of code and data, while compartments represent logical sets of cliques. HAKC optimizes for high performance inter-clique transitions, allowing developers to test compartmentalization policy designs that explore performance and security trade-offs. Additionally, HAKC allows developers to define an order of magnitude more compartments than past hardware enforced compartmentalization frameworks have traditionally allowed. HAKC minimizes developer effort by using a Clang LLVM compiler pass to inject instrumentation into source files. LKMs will be used as targets of HAKC implementation for two reasons. First, as revealed with the CVE data analysis (Figure 1 – 1), LKMs are the source of the majority of CVEs. Second, LKMs provide a logical boundary between Linux kernel subsystems. Kernel functionality related to a specific task, such as IPv6 protocol logic and packet filtering functionality are designated to separate modules.

The following sections will present the design and implementation of HAKC, as well as the application of HAKC for compartmentalizing the `nf_tables` LKM, a widely used packet filtering module.

Chapter 2

Background

The following sections outline the major attacks and vulnerabilities that motivate research in kernel vulnerability mitigation, as well as traditional efforts to thwart attacks that leverage such vulnerabilities through language based defenses, operating system design, and hardware based defenses.

2.1 Attacks and Traditional Defenses

Research in compartmentalization is generally motivated by attacks that have been discovered over time. The following sections broadly discuss general types of attacks and traditional defenses.

2.1.1 Code-injection Attacks

Such as C/C++ lack memory error checking functionality such as array bounds checking (more on safe/unsafe languages in Section 2.6). This gives rise to buffer-overflow attacks, which leverage common memory errors in unsafe languages to write data outside the bounds of an array. At one point, buffer-overflow attacks were capable of arbitrary code injection to alter the behavior of a program. For example, buffer-overflow attacks could leverage a lack of bounds checks on arrays to write data past the end of an array. If the array is located on the stack, then stack data can be corrupted or modified, allowing an attacker to inject code and run malicious code by modifying pointer values stored on the stack. Although code injection attacks pose serious threats to legacy software systems, the implementation of page protection schemes such as Data Execution Prevention (DEP) in x86 and Arm architectures [8, 9] and W$\oplus$X in operating systems page protections in operating systems such as Linux have mitigated such attacks by making data regions non-executable and disabling write permission for executable memory respectively. As we will see in the next section, buffer-overflow attacks are still relevant today because of there ability to aid in control-flow and data-oriented attacks.

2.1.2 Control-flow Attacks

Although code injection attacks have mostly been thwarted through DEP and W$\oplus$X memory permissions, control-flow attacks utilize *gadgets*, or preexisting code, to alter program execution. Though the attacker is restricted to only using gadgets to achieve a goal, most programs include libraries that significantly increase a program's exploitable attack surface. Control-flow attacks will leverage memory errors in unsafe languages such as C/C++ to modify function pointer values, *i.e.,* stack based return values, stack based jump values (Jump-oriented attacks), and heap based pointer values.

Various control-flow integrity (CFI) schemes have been developed to combat this style of attack by focusing on preserving the original call graph that was intended by the developer [10]. CFI solutions can vary in precision because of a lack of static information provided by a control-flow graph (CFG). For example, forward edges in a CFG represent all the functions that can be called by another function. These call instructions can include pointers to functions, which are computed during runtime. The possibility of forward edges can make forward edge CFI very imprecise. Backward edges in the (CFG) are easier to enforce with high precision because information on parent functions can be obtained during runtime. The use of shadow stacks is one example of CFI enforcement, which consists of storing metadata on a separate stack to detect when return pointers have been modified by an attacker [11]. Other more modern techniques for enforcing CFI involve cryptographically securing pointers (see Chapter 7).

Attacks such as control-flow attacks require knowledge of data and code memory locations. Traditionally, page-table hardware mechanisms were intended to enforce memory isolation at the granularity of individual pages. These isolation mechanism however has been broken with SPECTRE/Meltdown [12, 13] attacks, which utilize side-channel attacks and out-of-order execution to bypass CPU ring privilege checks and gain access to kernel memory. Memory location randomization techniques such as KASLR, which are intended to make data and code location discovery difficult, are subverted by these attacks and other side-channel attack. Kernel page-table isolation (KPTI, originally KAISER [3]) defends against Meltdown by maintaining both kernel side and user side page-tables, switching between these tables on a context switch. The user side page-table only contains kernel memory that should be accessed in user space (ring 3 privilege). Although KPTI enforces KASLR, KASLR is regarded as ineffective in protecting the operating system from an attacker that executes code in the kernel context [14]. This is a motivation for further exploration of kernel compartmentalization.

2.1.3 Data-oriented Attacks

Rather than modify function pointers, as is the case with control-flow attacks, data-oriented attacks focus on modifying data and data pointers. Works have shown that data-oriented attacks can leak sensitive user information in a similar manner to control-flow attacks [15]. One such example of real-world data-oriented attack is a Linux CVE-2016-4997 [16]. This vulnerability allows a user to provide a value to cleanup code in the IPv4 packet filtering system which can corrupt a value later used for pointer computation. The possible resulting damage of this exploit is any arbitrary integer being decremented, which could result in privilege escalation of a user process. This vulnerability, and data-oriented attack vulnerabilities in general are concerning because an attacker can change privileges and leak sensitive information, from *user space*, without violating the control flow intended by developers. An effective compartmentalization policy would limit the memory accessible by such an exploit, likely preventing damage such as unwarranted privilege escalation.

2.2 Compartmentalization by Operating System Design

The rise of complex attacks such as data-oriented attacks has motivated researches to explore compartmentalization as a simpler and more robust method of mitigating control-flow/data-oriented attacks. As will be discussed in the following sections, some such efforts include total operating system redesign.

2.2.1 The Spectrum of Operating System Compartmentalization

Kernel compartmentalization can be thought of as a spectrum of isolation granularity (Figure 2 – 1), where on one extreme exists monolithic kernels with minimal isolation, and on the other extreme exists zero-kernel operating systems, which utilize advancements in tagged architectures to achieve extremely fine-grained compartmentalization between components [17]. The following sections provide greater detail of several unique operating system architectures with varying degrees of compartmentalization and mechanisms for mitigating security vulnerabilities.

2.2.2 Monolithic Kernels

As discussed earlier, the monolithic kernel design has been adopted by several prevalent operating systems, including Linux based operating systems and Windows. Since the kernel in these systems has access to all memory, optimization can be made in terms of performance and ease of development. Unlike a microkernel architecture which typically requires 4 (and sometimes more) context switches on a system call, system calls in a monolithic kernel can take as little as two context switches. This results in better performance. Furthermore, one can argue that the use of shared

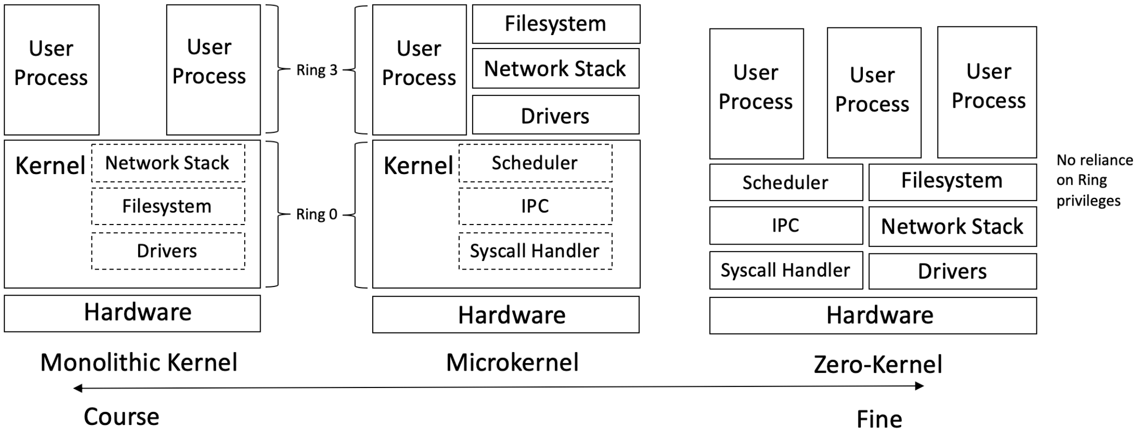


Figure 2-1: Compartmentalization as a spectrum of granularity. Notice that monolithic kernel and microkernel designs utilize Ring privileges to enforce isolation between user space and kernel space. The zero-kernel does not rely on Ring privileges for isolation, as memory is compartmentalized to the required granularity to enforce necessary isolation.

global data structures in monolithic kernels allows for easier development, as any piece of code in the kernel can access any necessary data structure. These design choice priorities result in minimal isolation between unrelated components such as drivers and schedulers. Often times, global access to kernel heap memory creates opportunities for attackers to access memory regions unrelated to the source of the security vulnerability [18]. As mentioned before, there exist hardware mechanisms such as SMEP/SMAP and page-table privilege checking to enforce memory isolation between the user-to-user and kernel-to-user logical boundaries at the granularity of pages. However, these mechanisms do nothing to enforce isolation within the kernel. As a result, a vulnerability in the kernel can compromise the memory of the entire system, including that of all user processes.

2.2.3 Microkernels

Microkernels such as Minix 3 [19] and the L4 family of operating systems (include SeL4 [20]) attempt to improve compartmentalization by moving kernel logic, traditionally found in monolithic systems, to user space. This provides the advantages of isolating bug-prone logic such as driver code as well as reducing the size of the kernel, which is considered the trusted computing base (TCB). This results in microkernels having significantly less trusted code (code operating in ring 0) than monolithic kernels - Minix 3 only has 4,000 lines of code while Linux has about 2.5 million lines of code. Consequentially, there is a significant decrease in the total number of expected kernel bugs [21]. The benefits realized from a smaller trusted computing based come with the cost of high inter-process communication (IPC). For example, consider RedoxOS [22], a developing microkernel written in Rust [23]. If a Redox user process is writing to a file, several context-switches need to occur. First, a context-switch from user to kernel space occurs. In kernel space, the destination for the system call is processed and the appropriate module is determined. Next, data is passed from the kernel to the filesystem module, result in another context-switch from kernel to user space. Once the fileystem has completed, either writing to an in-memory buffer or invoking two more context-switches to write to disk, control is passed back to the kernel, and then to the user, resulting in a total of at least 4 but (possibly 6) context switches. Efforts have been made to reduce the overhead of a context-switch through virtualized compartmentalization efforts, but these efforts generally involve the implementation of a hypervisor (see Section 2.2.4).

2.2.4 Exokernels and Unikernels

The concept of the Exokernel was introduced in 1995 as a re-definition of traditional operating system abstractions [24]. Rather than supporting filesystem, driver, and networking functionality in a privileged kernel space, as monolithic systems like Linux do, Exokernel leaves the implementation of such functionalities to applications. Applications then interface with a smaller and simplified "kernel" that multiplexes hardware interfaces such as disks and networking cards. This concept rooted in the common *end-to-end argument* in system design, where functionality is minimized in a low-level interface to increase flexibility for high level users of that interface.

The Exokernel abstraction forces traditional operating system functionalities, such as virtual address managing and IPC, to be implemented as libraries in applications. This has several major benefits. First, the Exokernel TCB is significantly smaller than monolithic systems because only the multiplexing of hardware resources needs to be trusted. Second, applications now have the flexibility to tailor traditional operating abstractions to there own needs. For example, applications that run independently on a machine may only require simple virtual memory abstractions if the application is focused on performance over security.

Inspired by the Exokernel computing environment, research efforts have proposed Unikernels as a modern operating system design to improve performance, security, and simplicity in cloud computing and embedded contexts [25, 26, 27, 28, 29, 30, 31]. Unlike the term Exokernel, the term Unikernel refers to the integration of LibraryOS's with applications. In a Unikernel based environment, the task of multiplexing hardware is placed on a hypervisor. Unikernel designs are particularly interesting in there cloud computing applications, as cloud computing applications generally focus on singular tasks (*i.e.,* a webserver or database). One such example of an implemented Unikernel design is MirageOS [28]. Implemented in OCaml, a memory safe language, MirageOS moves traditional operating system functionality like networking and filesystem management to OCaml implementations that are compiled as libraries and linked into application images. This results in a simpler computing environment with a smaller attack surface, as only necessary kernel code is compiled into the application image, while maintaining adequate performance. Furthermore, small image sizes allow for rapid rebooting, which is a direct advantage over virtual machines that manage entire traditional operating systems. Although efforts have been made to convert widely used operating systems to Unikernel structures, common industry reliance on monolithic computing stacks has made adoption slow.

2.2.5 Hypervisors/Virtualization based Microkernels

Similar to microkernel designs, efforts have been made to compartmentalize operating systems by leveraging hypervisors to isolate kernel subsystems [32, 33, 34, 35]. VirtuOS [32] is one example of a hypervisor based operating system isolation scheme that utilizes Xen to isolate kernel subsystems into separate service domains. VirtuOS uses less context switches during inter-domain communication when compared to traditional microkernel implementations. This is due to VirtuOS's implementation of exceptionless systems calls, which provide direct communication avenues between domains. Despite this optimization, inter-domain communication overhead is incurred from inter-domain data copying, file descriptor translation, and migration of other domain-specific information. Fault isolation is achieved between domains by using Xen to isolate domains, where each domain is designated a slim copy of a kernel binary. One unfortunate downside to VirtuOS, and other hypervisor based operating system compartmentalization schemes, is the increase in TCB, as the hypervisor becomes another trusted component in the software stack. This is a motivation for exploration into alternative compartmentalization schemes that do not increase the TCB.

2.2.6 Zero-kernel Operating Systems

On the far end of the compartmentalization spectrum, Zero-kernels rely on advancements in hardware isolation support to compartmentalize every facet of the traditional monolithic kernel. Tagged SoC architectures can be used to enforce control-flow and data access security policies at the granularity of individual words. Dover was used to compartmentalize ZKOS, a zero-kernel operating system in which every component of the traditional kernel is compartmentalized to a finer granularity than that of which can be achieved by monolithic and microkernel architectures [17]. However, architectures that provide such rich tagging schemes are Dover are unlikely to be adopted by industry any time soon, as most work has focused on tagging memory with limited tag bits.

2.3 Embedded Systems Compartmentalization

To minimize power and space consumption, embedded architectures traditionally lack virtual addressing capabilities. Often times the operating system and application are compiled into one image and share an address space. The prevalence of IoT devices have raised concerns over the lack of isolation and privilege separation for software running on embedded devices. Efforts have been made to provide inter-process as well as intra-process isolation using tagging extensions [36] as well as the Memory Protection Unit (MPU) commonly found in embedded devices [37, 38]. Other efforts target the lack of Control Flow Integrity (CFI) on embedded devices through a hardware performant extensions [39].

2.4 Formal Verification

As discussed earlier, common developer errors such as exceeding the bounds of an array, improper integer range checking, and use of null-pointers are not only a source of bugs but also a source of potential attack exploits. Inspection and rigorous testing have been used to locate software errors, especially as code develops. However, these methods are only as rigorous as the inputs to the system and, consequentially, the code paths that are tested.

Formal verification of software systems targets this problem by mathematically proving the correctness of the system against a formal specification defined by a developer. A system that is formally proven to be correct adheres to the formal specification of the developer and is free from possible developer induced bugs such buffer-overflow issues and lack of integer bounds checking.

Given the importance of security in operating system code, there has been research in applying formal verification methods to operating system development. One such example is SeL4 [20], a microkernel based off the L4 microkernel. SeL4 is unique in that the core kernel code is mathematically proven to be correct.

Unfortunately, the security benefits of formal verification of code comes with substantial costs. Firstly, formal verification requires immense computation power, making correctness proofs for large code bases infeasible. SeL4's small design of less than 10,000 loc, thanks to a microkernel architecture, makes formal verification on the kernel core feasible. However, formal verification of larges systems such as Linux, consisting of 25 million loc, is computationally infeasible. Furthermore, formal verification only guarantees correctness against a formally defined specification by a developer. Errors in the formal specification directly translate to errors in the code base. Additionally, unlike core kernel code, application specific extensions to SeL4 are not formally verified, increasing the attack surface of a system.

2.5 Symbolic and Concolic Execution

Traditionally programs are tested against test suites that apply specially selected inputs to uncover bugs in functions and systems in general. Since developers select inputs to test with, the quality of testing is constrained by the quality of inputs chosen by a developer. With this common practice of testing comes the risk of failing to check inputs that can result in unexpected security vulnerabilities, *i.e.,* buffer-overflow.

Symbolic execution is the systematic abstract analysis of code to ascertain inputs that will result in a program error. Unlike traditional testing, which uses concrete values to test a program, symbolic executors model inputs as abstract symbols. The symbolic executor than executes every possible code path, applying constraints to symbolic inputs as branches are encountered. Once a code path terminates, an SMT solver is used to calculate possible values for inputs such that the constraints on those inputs are satisfied, allowing a developer to test an error prone code path using the concrete values provided by the symbolic executor.

Unfortunately symbolic executors cannot be used on every program, as some programs have a computationally infeasible number of code-paths. This has resulted in the exploration of Concolic executors, which replace certain symbolic values with concrete values to avoid exploring infeasible code-paths [40, 41, 42]. For large code bases such as Linux, symbolic/concolic executors can leave large parts of the code base relatively unexplored, making language based and compartmentalization security schemes contenders in mitigating vulnerabilities and defending against vulnerability exploitation.

2.6 Language Based Defenses

Another approach to defending against malicious attacks is the use of a safe languages to mitigate memory errors. The following sections outline the difference between safe and unsafe languages and the benefits and costs associated with each.

2.6.1 Memory Safety

Memory safety refers to the restrictions a language places on a developer in order to minimize accidental memory accesses that deviate from the developers intentions. Memory Safety can be broken down into two components: Spatial Memory Safety and Temporal Memory Safety.

Spatial Memory Safety refers to restrictions a language places on a developer to mitigate accidental or intentional memory reads and writes to locations outside the bounds of the developers specifications. For example, when a developer allocates an array, the developer does not intend for data to be written outside the bounds of that array. Languages, such as Go [43], enforce a developers intentions such as bounded array reading and writing by adding runtime checks to array reads and writes. Reads and writes of an array that exceed the bounds of an array will result in a runtime error. Of course, these runtime checks come with a performance cost. Languages also implement static type checking to enforce spatial memory safety. A compiler will ensure that objects are treated as there original instantiation types unless otherwise specified by a developer (*i.e.,* through casting). Static type checking ensures developers do not accidentally mistake the types of objects, preventing data from being corrupted and written outside the bounds of the object. This advantage comes with the cost of less flexibility for the developer. The use of pointers to access arbitrary regions of memory violate static type safety rules, and must be explicitly stated by the developer. This often results in pieces of code being wrapped in `unsafe` blocks, which void all language memory safety guarantees.

Unlike spatial memory safety, temporal memory safety focuses on the *time* at which a region of memory is used. More specifically, languages that enforce memory safety mitigate accidental re-use of an object after the object as been reallocated for a different purpose. Temporal memory safety is therefore primarily concerned with heap based memory, as developers continuously allocate/free/reallocate heap memory at runtime. Oftentimes languages such as Java will provide a garbage collection system that monitors the use of objects, only freeing objects when there are no longer references to an object. This relieves developers from the burden of tracking the lifetime of memory, but comes at the cost of increased performance overhead during runtime.

2.6.2 Safe and Unsafe Languages

Languages such as C/C++ are extremely popular choices for system implementations because they provide flexibility in terms of memory access and performance in terms of speed. Despite these benefits, these languages do not implement the static or dynamic spatial and temporal memory safety checks other languages perform, exposing systems implemented in these languages to a plethora of developer errors and security vulnerabilities [44].

Unlike unsafe languages, safe languages provide the statically and dynamically enforced memory safety mechanisms. For example, Go is a statically type checked language that provides a garbage collector to enforce temporal memory safety. Although garbage collectors are the general mechanism safe languages leverage to enforce temporal safety, some system languages like Rust [23] are able to statically enforce temporal memory safety at compile time, removing the need for a costly runtime garbage collector. Rust ascertains lifetime information by enforcing a strict ownership model and, in some cases, requiring that developers specify the lifetime of an object [23]. When used for low-level systems development, safe languages provide an `unsafe` specifier that can be used to subvert memory safety restrictions. The `unsafe` specifier is generally used when accessing memory in an unsafe manner (*i.e.,* reading input from memory mapped devices or operating system context-switching). To gain the security benefits of safe languages, developers should minimize the use of unsafe languages. However, in practice, unsafe blocks have been shown to be used extensively [45].

2.6.3 Operating System Safe Language Implementations

At the level of language selection for operating system implementation, we have seen operating systems written in memory and type safe languages to help eliminate vulnerabilities. Examples include BiscuitOS, a monolithic kernel written in Go [46], RedoxOS [22] and TockOS [47], microkernels written in Rust, and Singularity, a microkernel variant written in Sing# [48].

Singularity is an operating system written by Microsoft Research and released in 2007. The goals of the project were to utilize a type safe and memory safe language to write a trustworthy operating system. Although Singularity is technically classified as a microkernel, the system differs in that all processes (including the kernel) share the same address space. Memory safety is enforced with the concept of a Software-Isolated Process (SIP), which utilizes garbage collection and static verification to guarantee that pointers cannot accidentally access memory outside of granted regions. Additionally, communication between SIPs is secured through contract-based channels that are enforced by the Sing# language and allow applications to agree on a verifiable communication method.

Unfortunately, the aforementioned operating systems come with significant costs. Written in a high-level language with a garbage collector, BiscuitOS experiences slowdowns of 5-15% when compared to Linux [46]. Singularity provides improved security of modern kernels at the cost of a dramatic redesign that is infeasible to implement for large existing kernels such as Linux. Although these systems are mostly implemented with safe languages, they still contain bits of unsafe code that are required for low-level memory and register manipulation. The existence of unsafe code and costly adoption is yet another motivation for exploring kernel compartmentalization.

2.7 Compartmentalization with Hardware

Most of what has been discussed thus far relates to software defenses against malicious attacks. The following section explore innovations in hardware defenses that enforce isolation and security policies. See Chapter 7 for a more extensive overview of modern hardware based compartmentalization efforts.

2.7.1 Virtual Addressing

Almost all conventional modern CPUs support virtual addressing schemes that allow the operating system to manage process access to memory at a granularity of page (typically 4KB in size). Virtual addressing is the main mechanism utilized by monolithic operating systems to enforce user-to-user and user-to-kernel isolation. Memory privilege enforcement, normally implemented as Page Table Entry (PTE) flags, also give operating systems flexibility in determining what kind of permissions to grant a process at the granularity of pages. This allows operating systems to enforce W⊕X permissions and DEP on code regions in a virtual address space. Additionally, microkernels leverage this hardware functionality to maintain separation between domains. Other work has focused on using the isolation benefits of virtual addressing to move driver implementations to user space for Linux subsystems. Sud [49] moves devices drivers to user space and relies Linux kernel virtualization in user space to prevent malicious drivers from accessing memory outside their defined scope.

2.7.2 Protection Ring Model

The Protection Ring Model is a privilege separation mechanism implemented in most conventional CPUs that categorizes instruction privileges in 3 or 4 rings. Operating systems typically execute instructions in Ring 0 as this is the most privileged ring, allowing full access to all instructions, memory, and peripheral devices. User process normally execute in Ring 3, the least privileged ring. Normal monolithic operating systems generally operate in two rings, using Ring 0 for kernel execution and Ring 3 for user process execution. Recent CPUs from AMD and Intel that support hypervisor virtualization now offer Ring -1. This allows guest operating systems to operate in Ring 0, independently from one another.

2.7.3 Arm PAC and MTE

Unlike the previously discussed hardware tagging extensions, Arm has also released a Memory Tagging Extension (MTE) [7] and Pointer Authentication Codes (PAC) [6] extension profiles for the ArmV8-8.5 architecture. These extensions will be implemented in hardware, making them particularly attractive for compartmentalization implementation. HAKC relies on these extensions and will be discussed in more depth in upcoming chapters.

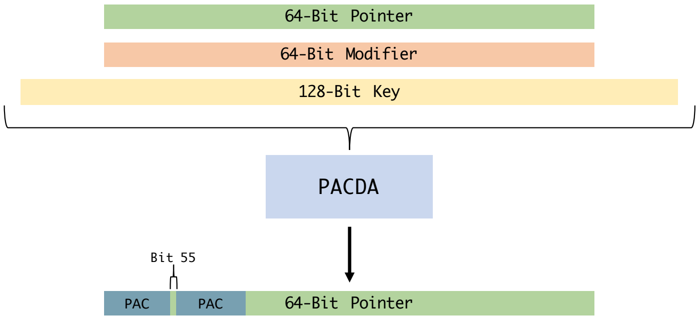
Chapter 3

Design

The following sections present the design of Hardware-Assisted Kernel Compartments, HAKC, a kernel compartmentalization framework that leverages Arm PAC and MTE hardware extensions as well as Clang LLVM analysis and instrumentation to enforce a two-tier compartmentalization scheme.

3.1 High Level Design Decisions

As discussed earlier, HAKC is intended to compartmentalize the Linux kernel, focusing on, but not limited to, LKM compartmentalization. HAKC supports a two-tier compartmentalization scheme consisting of *cliques* and *compartments*. Each clique is a unique logical grouping of data and code that resides within a compartment. HAKC is agnostic towards compartmentalization granularity, allowing developers to test both course-grained and fine-grained compartmentalization schemes. HAKC enables developers to regulate inter-compartment and inter-clique data and control flow by providing an API developers can use to define flow policies. The following sections describe both relevant Arm extensions and HAKC design in greater detail.



(a) A pointer, modifier, and 128-bit key are used to compute a PAC that is stored in the upper bits of the resulting pointer. Bit 55 determines whether the top or bottom half of the address space is used.

Extension	Description
PAC<*key*>	Signing instruction
AUT<*key*>	Authentication instruction
IA,IB	Instruction Keys
DA,DB	Data Keys
GA	General Key

(b) PAC relevant architectural extensions

Figure 3-1: PAC implementation and generation

3.2 Arm PAC and MTE

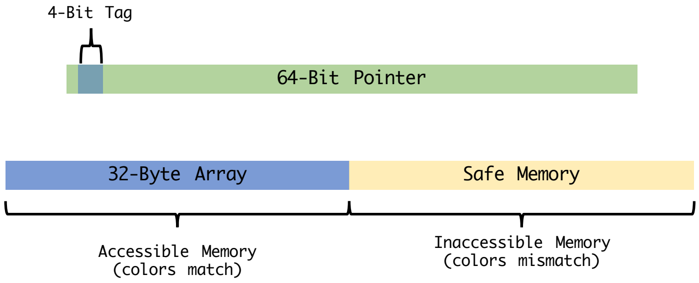
The following sections describe the relevant details of Arm PAC and MTE hardware security primitive implementations.

3.2.1 Arm Pointer Authentication Code (PAC) Extension

PAC is an is an Armv8-A hardware extension that allows software to cryptographically harden the integrity of a pointer value. PAC leverages the fact that upper bits of pointer values are generally unused for addressing, and can be utilized to store pointer metadata information.

Figure 3 − 1a outlines the basic usage of PAC to sign a pointer. The PAC architectural extension provides several new instructions that can be categorized into a signing group and authenticating group (Figure 3 − 1b). During signing, a pointer and modifier are passed as input to a specified PAC<*key*> instruction. The resulting PAC is stored in the upper bits of the pointer. During authentication, an AUT<*key*> instruction is used to compute a new PAC given a pointer, modifier and *key*. If the computed PAC does not equal the PAC stored in the upper bits of the pointer, an AUT<*key*> returns an invalid pointer.

PACs are computed using a cryptographically secure hardware implemented QARMA block cipher [50]. The PAC size can range from 3 to 31 bits depending on the size



(a) The 4-bit pointer tagged is compared against the 4-bit memory tag. Unequal tags incur a memory fault. One can think of tagging memory as "coloring", where a 4-bit tag yields 16 colors

Extension	Description
STG	Tag store instruction
ST2G	2 granule tag store instruction
LDG	Tag load instruction

(b) MTE relevant architectural extensions

Figure 3-2: MTE implementation

of the virtual address space used (which ranges from 32 to 52 bits), and if memory tagging is enabled (which uses 1 byte, more on this in Section 3.2.2). As shown in Figure 3 – 1b, PAC provides 5 different 128-bit keys that can be used during PAC signing/authentication.

3.2.2 Arm Memory Tagging Extension (MTE)

MTE is designed to allow developers to improve the spatial memory safety of software by tagging every 16-bytes of memory with a 4-bit tag. Using the top-byte-ignore (TBI) architectural feature, a 4-bit tag is also stored in the top byte of a pointer. We can refer to these tags as "colors", where each memory and pointer are colored from a possible selection of 16 different colors. When a load or store is performed, MTE functionality will check if the pointer color and memory color match, throwing a memory error upon mismatch. As shown in Figure 3 – 2a, this scheme can be used to enforce memory safety. For example, one can color memory after the end of an array, preventing attack methods such as buffer-overflow. MTE provides several instructions to load and store tags at provided memory addresses (Figure 3 – 2b).

Unfortunately using MTE's tagging scheme only allows for up to 16 unique compartments in any compartmentalization scheme. For large systems such as Linux, a larger number of compartments would yield better security. HAKC proposes solution that leverages MTE to provide $4 * 10^{15}$ compartments.

3.3 Threat Model

The attacker is assumed to not have root access in the system, and is therefore unable to modify kernel modules. However, the attacker is able to invoke arbitrary system calls and can send arbitrary data to the kernel (for example via `netfilter`, which supports socket communication between user space and kernel modules). Furthermore, LKMs are not considered to be malicious, but only exploitable through security vulnerabilities. All non-compartmentalized kernel code is considered trusted, as verifying the integrity of non-compartmentalized kernel code and data passed to LKMs is considered computationally infeasible (see Section 2.4 for prior work on such verification problems). The Arm core SoC is also part of the trusted computing base with several exceptions, IO devices, direct memory access (DMA), and side-channel attacks such as SPECTRE [12] and Meltdown [13].

3.4 Compartmentalization Scheme

As stated earlier, HAKC allows developers to define a two-tiered compartmentalization scheme consisting of cliques and compartments. To allow developers to easily test different compartmentalization configurations, HAKC provides a developer API that defines logical grouping and flow policies as follows:

1. *Clique*, a logical grouping of code and data that is associated with a single color. There is maximum of 16 possible cliques.

2. *Compartment*, a logical grouping of one or more cliques, associated with a compartment *ID* (2^{48} possible values). All data within a compartment is owned by exactly one clique.

3. *Clique Access Policy*, control-flow and data-flow policy defining what cliques have access to the current clique. This is a binary value indicating if one clique can transfer control-flow to another clique.

4. *Compartment Access Policy*, control-flow and data-flow policy defining the entry and exit cliques for a given compartment, as well as which compartments have access to the given compartment.

Furthermore, we define the following terms to capture compartmentalization relevant events during execution:

1. *ownership*, the logical assignment of data and code to a singular clique. Ownership can be transferred.

2. *control flow*, the execution of instructions. Includes direct and indirect calls.

3. *data flow*, the access of data by load and store instructions.

4. *inter-clique transition*, when control flow and data flow occur over the boundary of two cliques. This can occur as a result of a load, store, direct, or indirect function call. Clique access policies are enforced during an inter-clique transition.

5. *inter-compartment transition*, when control flow and data flow occur over the boundary of two compartments. This can occur as a result of a direct or indirect function call. Compartment access policies are enforced during an inter-compartment transition. Ownership of data and code is transferred upon inter-compartment transitions.

To aid in describing various compartmentalization policies, let's define some helpful notations. Let $C_{i,j}$ refer to a clique with color i and compartment j. Let $P_{i,j}$ be a set of colors representing the access policy for $C_{i,j}$. Let A_j represent the compartment with id j, and T_j be a tuple of sets of entry and exit cliques that represents the access policy for A_j.

Figure 3 – 3 presents an example compartmentalization policy graph. The directed edges represent clique and compartment access policies, where the existence of an edge indicates that a source claque has granted access to a target claque. For example, $C_{y,3}$ would have an access policy $P_{y,3} = \{r, g\}$, because $C_{y,3}$ has edges directed towards

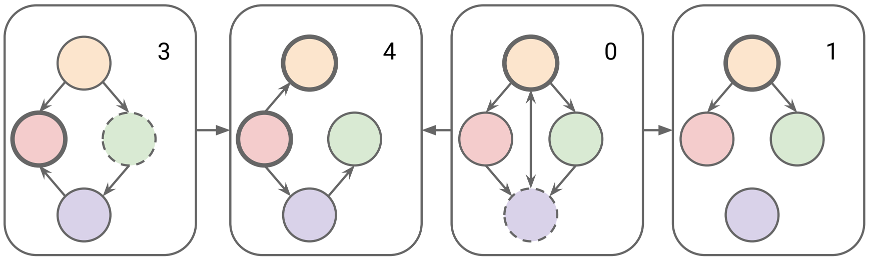


Figure 3-3: Compartmentalization policy graph. Outline boxes represent compartments, circles represent cliques. Directed edges represents clique and compartment access policies.

the *red* and *green* cliques. On the other hand, clique $C_{r,3}$ would have access policy $P_{r,3} = \emptyset$, as all edges point towards the *red* clique. Note in compartment A_1, there is a clique $C_{p,1}$ that is completed isolated, meaning there exists no access policy in A_1 that contains the color p. In compartment A_0, we see that it is possible for two cliques, $C_{y,0}$ and $C_{p,0}$, to grant mutual access to each other. Notice some cliques are compartment entrances (outlined in a bold line) while others are compartment exits (outlined in a dotted line).

3.5 Arm PAC and MTE Compartmentalization Enforcement

This sections discusses the design decisions made to utilizing PAC and MTE to support and enforce compartmentalization.

3.5.1 Compartmentalization Data Storage

To properly enforce a secure compartmentalization scheme, clique colors, compartment IDs, and access policy information must be stored properly.

The top-byte-ignore architectural feature (TBI) is used to store compartment I.D.s in the top byte of a pointer. Colors are stored as MTE tags. Each clique contains meta-data that represents policies defined by the developer (Figure 3 – 4).

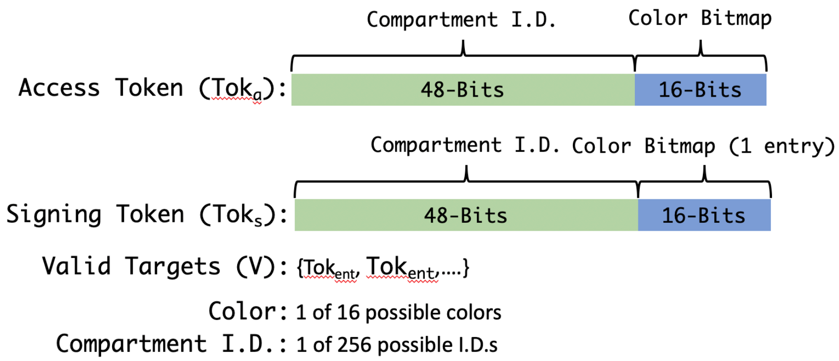


Figure 3-4: This is per-clique meta-data that is defined by a developer

3.5.2 Compartmentalization Enforcement

In general, pointers are signed using PAC (see Figure 3 − 1a). For a given clique C_{r_3}, a singing token Tok_s (see Figure 3 − 4) is used as a modifier. This signing token is formed by concatenating a 48-bit string representing the compartment I.D. (3 in the case of C_{r_3}) with a 16-bit bitmap, where the index for color r is set to a 1, and all other values are set to 0. This signing token effectively acts as a unique *context* for signing the pointer. Pointers located in different compartments/cliques would be signed with a modifier representing a different context. Currently, the compartment I.D., 3, is then stored in the top byte of the pointer (thanks to TBI). Since all cliques within the same compartment share a compartment I.D., the compartment I.D. can be stored in other places as well without inducing a significant performance cost. Using MTE instructions, the object pointed to by the pointer is then colored r. The color and compartment I.D. will be used for authentication in the future. A PAC signing instruction is then used to embed a PAC in the upper bits of the pointer.

Upon pointer authentication, the contextual modifier is recreated by concatenating the compartment I.D. stored in the top byte of the pointer with the color of the object of the pointer. A bitwise-or operation is then performed between the modifier and access token Tok_a. In our running example, if Tok_a does not contain a value of 1 at index r, or if the compartment I.D. of Tok_a differs from that used when signing, 3, the contextual modifier will not match the signing Tok_s, resulting in a failed authentication. The pointer and contextual modifier are then passed to a PAC authentication instruction, which verifies the PAC stored in the pointer. If the verification fails, the pointer is tainted, resulting in a memory fault upon future dereference. If an attacker changed the value of either the compartment I.D. or color, the attacker would need to guess the value of the PAC. In the case of HAKC, a 16-bit PAC is used, so an attacker would have a $1/2^{16}$ chance of guessing the correct PAC. If the attacker fails to guess the correct PAC (which is highly probable), then pointer authentication fails.

In order to ensure that control-flow and data-flow transitions abide to clique and compartment access policies, access polices must be enforced for every unique store, load, direct, and indirect call instructions. In C, this translates to every unique pointer dereference, as well as every function call. Optimizations are made using an LLVM compiler pass to avoid unnecessary policy checking. The following paragraphs describe instruction specific details that are relevant to enforce control and data flow.

Call Instructions: Global values and stack allocated structs that are passed as parameters to any function are signed. Call instructions are handled in different manners depending on an inter-clique transition or an inter-compartment transition.

During an inter-clique transition, indirect call pointers are authenticated in the general manner, where a compartment I.D. and color are retrieved to authenticate a pointer against a contextual modifier.

If authentication fails for an indirect call, the indirect pointer is treated as an inter-compartment call. Each clique has access to a *valid target* list V. V contains all the possible compartments that a clique's compartment is permitted to access. Iterating through V, the indirect call pointer is authenticated against each possible contextual modifier. If an authentication check passes, the call is permitted. Parameters passed during inter-compartment calls need to be resigned for the new compartment/clique context. This results in transfer ownership of the data to the new compartment. Upon return, data ownership is transferred back to the context of the caller. For inter-compartment direct calls, only data ownership transferring is performed.

Loads and Stores: In order to ensure proper data-flow access policy enforcement, policy checks must be made on loads and stores to memory addresses. These checks involve general authentication as described earlier, where a contextual modifier is generated and certified by the clique access control token.

Data pointer authentication checks occur at the earliest moment in the execution of an LLVM basic block. Special care is taken to assure that pointers are authenticated before dereferences and pointer arithmetic occur. LLVM instrumentation is explained in greater detail in Chapter 4.

3.6 Benefits of a Two-tiered Compartmentalization Scheme

One could imagine a design in which the two-tiered compartmentalization scheme is flattened, effectively removing the logical construction of a compartment. This alternative design would only consist of cliques, where each clique is indexed by unique id/color combination. Unlike the two-tiered design, a flattened scheme would necessitate that claque access policies encode the access policy of all possible cliques, which can grow to be an infeasible amount of cliques. Although this hasn't been tested, we can confidently predict that checking a data structure that encodes this amount of information would be orders of magnitude more expensive than checking a single 64-bit integer. By having a two-tier compartmentalization scheme, clique access policies only need to encode the access policy for 16 claques. In this way, the issue of costly access policy data structures is moved from inter-claque flow to inter-compartment flow (by needing to check a valid targets list). This is a conscious trade-off made to optimize for inter-clique flow. Having inter-clique control and data flow perform faster than inter-compartment control and data flow also allows developers to devise various compartmentalization schemes that trade off performance and security. For example, a scheme with minimal compartments will perform faster than a scheme with more compartments, but will not be as secure as a scheme with maximal compartments. This is directly tied to the resigning of data upon ownership transfer discussed earlier. Although performance intensive, data resigning makes malicious inter-compartment data flow more challenging.

Chapter 4

Implementation

This chapter covers the implementation details of HAKC and HAKC compartmentilzation scheme for the `nf_tables` LKM.

4.1 Developer Instrumentation

The current implementation requires developers to add source code instrumentation to both specify compartmentalziation policies and apply various ownership transfers that the LLVM compiler pass is unable to infer.

4.1.1 Policy Definition Instrumentation

As discussed earlier, developers have fine-grained control over the scope of cliques, compartments, clique access policies, and compartment access policies. Currently, the HAKC LLVM pass requires developers to specify these logical parameters in source code. HAKC provides an API for developers to easily instrument regions of code with defined policies.

Figure 4—1 presents an example developer use of the API to used compartmentalize a file in the `nf_tables` module. The HAKC API provides macros that can be used to specify clique color, compartment I.D., and whether the current clique can access other compartments. These macros generate data structures that are then used during

```
// code snippet from net/netfilter/nf_tables_api.c
#include <linux/hakc.h>
#if IS_ENABLED(CONFIG_PAC_MTE_COMPART_NF_TABLES)
#include <linux/hakc-transfer.h>
HAKC_MODULE_CLAQUE(3, BLUE_CLIQUE, HAKC_MASK_COLOR(SILVER_CLIQUE));
HAKC_EXIT(HAKC_ENTRY_TOKEN(0, HAKC_MASK_COLOR(SILVER_CLIQUE)),
              HAKC_ENTRY_TOKEN(1, HAKC_MASK_COLOR(SILVER_CLIQUE)));
#endif
```

Figure 4-1: Developer policy definition instrumentation

```
// code snippet from net/netfliter/nf_tables.c
// transfer function example
#if IS_ENABLED(CONFIG_PAC_MTE_COMPART_NF_TABLES)
DEFINE_HAKC_OUTSIDE_TRANSFER_FUNC(nf_tables_gettable, int,
                 struct net *net, struct sock *nlsk,
                 struct sk_buff *skb, const struct nlmsghdr *nlh,
                 const struct nlattr * const nla[],
                 struct netlink_ext_ack *extack)
{
    net = hakc_transfer_to_clique(net, sizeof(*net), __claque_id,
   __color, false);
    skb = hakc_transfer_skb(skb, __claque_id, __color);
    nlh = hakc_transfer_to_clique((void*)nlh, nlh->nlmsg_len,
   __claque_id, __color, false);
    return nf_tables_gettable(net, nlsk, skb, nlh, nla, extack);
}
#endif
```

Figure 4-2: Developer manual transfer function

signing and authentication. In this case, the current clique can access blue and silver cliques within the same compartment, and can access the silver entry cliques of compartments 0 and 1.

4.1.2 Ownership Transfer Instrumentation

Along with policy specification, developers are required to provide transfer instrumentation when compartment boundaries cannot be inferred. Figure 4 − 2 is an example piece of code that manually transfers `nf_tables_gettable`, a function that is called by the kernel, outside the module. Since uninstrumented kernel code is consider part of the trusted computing base, developers must manually transfer all data that is passed as parameters into the function. Developers are currently also required to manually transfer kernel struct allocation. Some of these requirements will be relaxed in future automated versions of HAKC (see Chapter 6).

4.2 LLVM Instrumentation

HAKC signing logic and authentication checks are added to source during compilation with the Clang/LLVM/C++ compiler pass. HAKC uses the Clang LLVM API to analyze and inject necessary instrumentation into source code. Source files with developer instrumentation are analyzed. Code and data sections that require coloring are indicated with special ELF file sections. Upon module load, these sections are colored accordingly. Traditional page-level protections are also applied to relevant code and data sections as per usual. Note, changing the color of a certain address does not require write permissions to that address. The LLVM pass is capable of even greater autonomy, relieving developers of most instrumentation. These future possible changes are discussed in greater detail in Chapter 6.

4.2.1 Inter-Procedural Optimizations

The Linux kernel makes extensive use of functions that are restricted to being used within one compilation unit, such as functions declared with the `static` keyword. We can leverage this fact to minimize pointer authentication checks. For example, if all caller functions to function F authenticate a parameter for function F, then that parameter can be passed as an authenticated value, and authentication for that parameter does not need to occur in F. LLVM allows inter-procedural analysis to be performed, where every function maintains a set of pointers, $P_{start,F}$, that are known to be authenticated at F's entry, as well as the set of pointers F authenticates, $P_{auth,F}$. P_{start} is initially the empty set for all functions, and let $P_F = P_{start,F} \cup P_{auth,F}$. At every call site to F, each pointer argument, p, is in the caller function's P, and if the pointer argument is in all P, then $P_{start,F} = P_{start,F} \cup p$. This analysis repeats until a steady state is achieved, and no P_{start} changes. For the LKMs we compartmentalized, this analysis reduced the number of data check insertions by 2%. The number of global variables that needed to be signed, because their addresses are passed to other compartmentalization functions, is reduced by 8%. These reductions translate to an average of 12% fewer data authentication checks and 19% fewer transfers during experimentation.

4.2.2 Intra-Procedural Optimizations

Another opportunity for optimization comes from the observation that pointer authentication checks do not need to occur until immediately before a pointer is dereferenced. For example, if a pointer *p* is only dereferenced inside a single branch of an if-statement, then *p* will only needs to be authenticated inside that if-statement. Authentication checks for a given pointer *p* can be placed at the closest parent block of all blocks that dereference *p*. In certain edge cases, limiting pointers to one authentication per function incurs overhead because two pointer dereferences may share a parent block, but neither of the dereferences are made. This can occur if a pointer is only dereferenced during error handling with a `goto` statement.

4.3 Linux Packet Filtering

Both `nf_tables` and `ipv6` LKMs were used to test and evaluate HAKC. The following sections will provide an in depth description of `nf_tables` and how it relates to the greater networking stack, and highlight the efforts in compartmentalizing the module.

4.3.1 Linux Packet Filtering

Linux provides various modules and functionality to support packet filtering (*i.e.,* `ipchains`, `iptables`, `eBPF`). For the purposes of HAKC testing, we have chosen to compartmentalize `nf_tables`, a packet filtering module that is intended to be a replacement for it's predecessor `iptables`. The following sections will first provide an overview of portions of the Linux networking stack relevant to packet filtering, then discusses `nf_tables` related architecture and HAKC compartmentalization efforts.

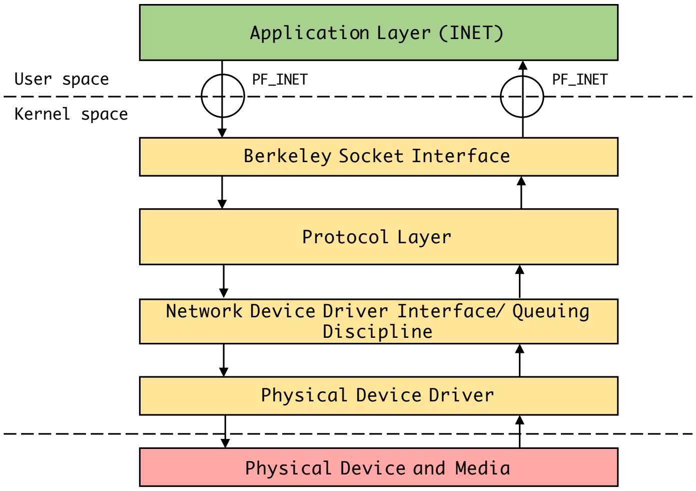

Application Layer (INET)
User space
Kernel space
PF_INET
PF_INET
Berkeley Socket Interface
Protocol Layer
Network Device Driver Interface/ Queuing Discipline
Physical Device Driver
Physical Device and Media

Figure 4-3: Diagram of the Linux networking stack architecture. User space applications communicate over Berkeley socket interface. Diagram taken from Affix documentation [51]

4.3.2 Brief Overview of the Linux Networking Stack

At a high level, the Linux networking stack is responsible for managing packet traffic flow through a machine. This includes packets received by applications, sent by applications, as well as packets that are traveling to other machines on the network (*forwarding*). Figure 4 – 3 presents a diagram of the Linux networking stack. The bulk of packet processing occurs in the Protocol layer in Figure 4 – 3) where protocols such as IPv4/IPv6 and TCP execute logic on incoming/outgoing packets. Applications communicate to the kernel through protocol specific sockets (INET protocol in the case of Figure 4 – 3). Linux makes extensive use of `struct sk_buff`, which is a representation of a packet. Incoming packets are processed by physical devices (such as a network card). Device drivers initialize `struct sk_buff` from packets. Pointers to `struct sk_buff` and then passed between networking layers.

4.3.3 Netfilter

`netfilter` is a layer of infrastructure that is embedded in the network stack and provides *hooks* kernel modules can use to attach functions in various places along packet routing paths. Figure 4 – 4 presents a diagram of the packet routing throughout

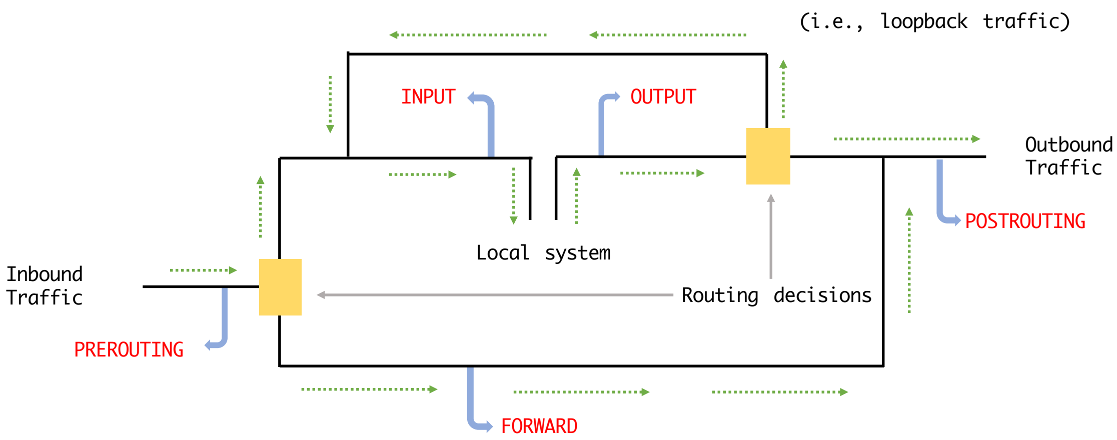


Figure 4-4: Diagram of `netfilter` hook locations with respect to routing. The green dotted arrows represent packet traffic. Hooks are labeled in red. Diagram taken from here [52]

.

the network stack. Each hook (labeled in red) allows kernel modules to access packets along different stages and paths of routing. `netfilter` handles packets based on hook function return values defined as follows [53]:

1. *NF_ACCEPT*: continue traversal as normal
2. *NF_DROP*: drop the packet
3. *NF_STOLEN*: Module taken over packet, don't continue traversal
4. *NF_QUEUE*: queue packet
5. *NF_REPEAT*: call hook again

`netfilter` hook functions are passed packets on function invocation, and are free to modify packet contents. This makes `netfilter` suitable for supporting functionality that leaves packets intact, like packet filtering, as well as functionality that modifies packets, like network address translation.

4.3.4 nf_tables

`nf_tables` is a packet filtering module that utilizes `netfilter` to provide packet filtering capabilities for user space processes. `nf_tables` provides an extensive API developers can use to filter and track packets at most netfilter hooks. The following list is a subset of the API that is relevant to evaluation discussed later:

1. *Tables:* nf_tables implement functionality that includes connection tracking, network address translation, and packet filtering. These tables consist of chains and rules that define functionality.

2. *Chains:* sets of rules that are evaluated on packets.

3. *Rules:* formally defined pieces of logic that determine output an action for a given packets attributes. Rules take as input packet attributes (*i.e.,* interface type, destination and source address, IP address values) and output a decision (*i.e., NF_ACCEPT* or *NF_DROP* in the case of packet filtering).

nf_tables also provides users with the additional abilities such as counting or logging packets, however this functionality was not tested during HAKC evaluation.

Through the netlink module, user space applications can communicate with the nf_tables API over sockets. The user is able to specify which hook in the routing path to attach a packet filter to, as well as the netfilter hook priority. nf_tables both implements packet filters and modifies packet filters in real-time.

nf_tables is a desirable candidate for HAKC implementation testing for several reasons. Firstly, nf_tables receives and processes potentially malicious user data when constructing data structures. This information includes table name strings and rule specifications. Secondly, nf_tables has direct access to critical packet information, where modification could affect both the local machine and other machines on the network.

Chapter 5

Evaluation

When evaluating HAKC, we wanted to answer the following research questions:

1. What is the overhead imposed by HAKC?

2. What is the overhead of using multiple compartments in a single system?

3. Will users notice any difference in performance under real-world work loads?

To answer these questions, we used HAKC to compartmentalize the `ipv6` and `nf_tables` LKMs. The following sections describe our experimental setup, benchmarks, and simulated user browsing data. We performed all evaluations on a Raspberry Pi 4 8GB, and our kernel version was based off the Debian 5.10.24 source.

5.1 Instruction Analogs

As of June 2021, no hardware implementing MTE is available, and the most readily available hardware implementing PAC are Apple devices containing the A12 processor, and are unfortunately heavily locked down. Therefore, in line with the evaluation methodology of Liljestrand, et al. [54], we ensure correct functionality using emulation, and measure overhead using instruction analogs. An instruction analog has the same number of CPU cycles and memory footprint as the PAC/MTE instruction it is intended to mimic, but does not perform security check. We use instruction analogs

```
ldg xT, xN
  ldr x16, [xN]
  mov x17, #0xF0
  lsl x17, x17, #49
  and xT, x17, x17
```

```
stg xT, xN, imm
  ldr x16, =TAG_MEM
  mov x17, xT
  lsr x17, x17, #49
  str xT, [x16]
```

Figure 5-1: Implementation of MTE instruction analogs

to accurately estimate performance overhead in lieu of the actual instructions. The PAC analogs are adapted from the PARTS system detailed in [54], and the following describes the implementation of MTE analogs.

Version 5.10 of the Linux kernel uses the load and store instructions for single or multiple tags, namely `ldg`, `stg`, `ldgm`, and `stgm` respectively. For the single tag instructions, the kernel only uses the post-index encoding. Figure 5 − 1 presents the MTE instruction analog implementations. To simulate the `ldg` instruction, which writes the tag to bits 49–53 of an input register, we perform a load of the target address, and finally place a valid tag value in the appropriate register bits. To store a tag, we retrieve the tag from bits 49–53 of the pointer address, and write to a global variable. Multiple tag operations simply repeat these single tag operations. We took care to ensure that memory accesses occur at every MTE instruction to simulate a worse case memory tag access, but an actual implementation of MTE could include tag caching or other performance enhancements. Thus, we claim that our performance overhead is an estimation of *worst case* performance.

5.2 Single Compartment Performance Overhead

The following results were obtained by compartmentalizing `ipv6` and `nf_tables` LKMs into single compartments. A host machine running ApacheBench [55] was used to benchmark a standard unmodified Apache web server running on a Raspberry Pi. Each test was performed 10 times on files of size 1KB, 1MB, and 10MB. Each set of figures presents the number of requests made per second and the amount of KB of data transferred per second. We measured all performance overhead relative to the unmodified kernel. Both kernels share the same user-space.

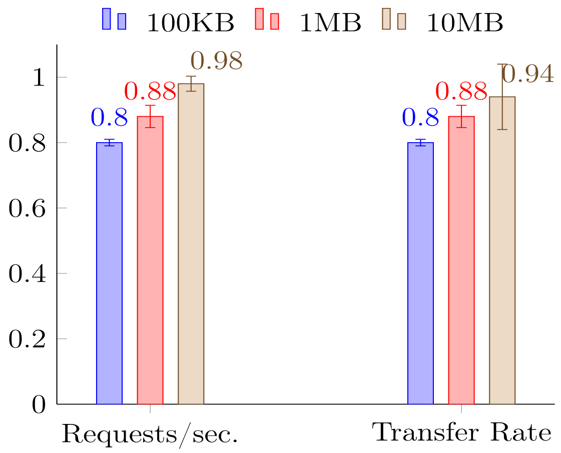


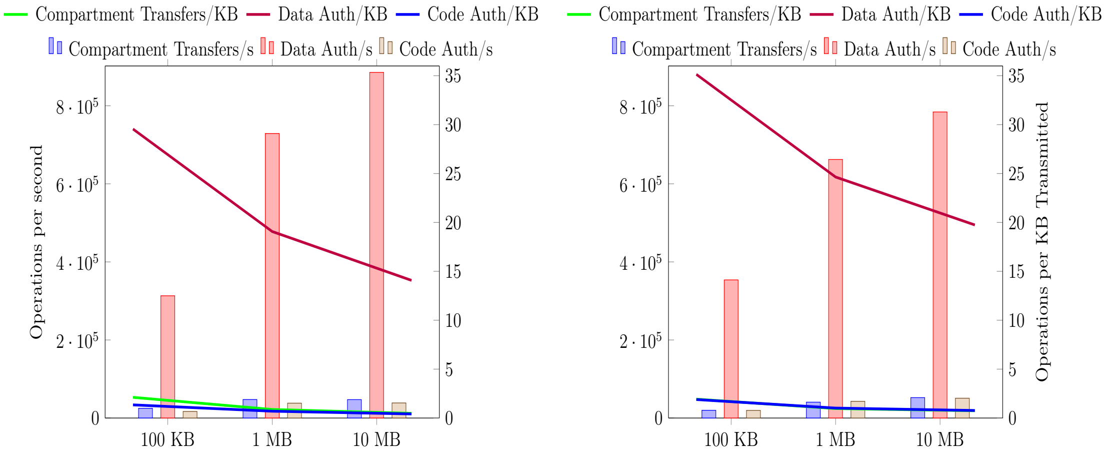
Figure 5-2: `ipv6` overhead normalized to unmodified kernel when transferring various sized payloads.

5.2.1 `ipv6`

`ipv6` was compartmentalized into one compartment consisting of two cliques that shared mutual access to each other's code and data.

The overhead measurements for `ipv6` are listed in Figure 5 − 2, normalized to the performance of the unmodified kernel. Overall, the performance of our `ipv6` compartmentalized LKM is good compared to the baseline, with only a 20% reduction in both requests per second and transfer rate in the worst case.

When the transfer size is small, the establishment of the TCP connection imposes significant overhead relative to the actual transferring process. Once the TCP connection has been established, however, relatively few data checks need to be performed to transfer the payload. This explains the low 2%–4% overhead for the 10MB payload measurement; larger payloads spend less time establishing the TCP connection relative to the total transfer time. Figure 5 − 3a shows this behavior in the HAKC operations per KB Apache sends. HAKC operations include the number of compartment transitions, the number of data pointer authentications, and the number of code pointer authentications. While the number of operations per second either increase or remain constant, the number of operations per KB of transmitted data monotonically decreases with payload size.



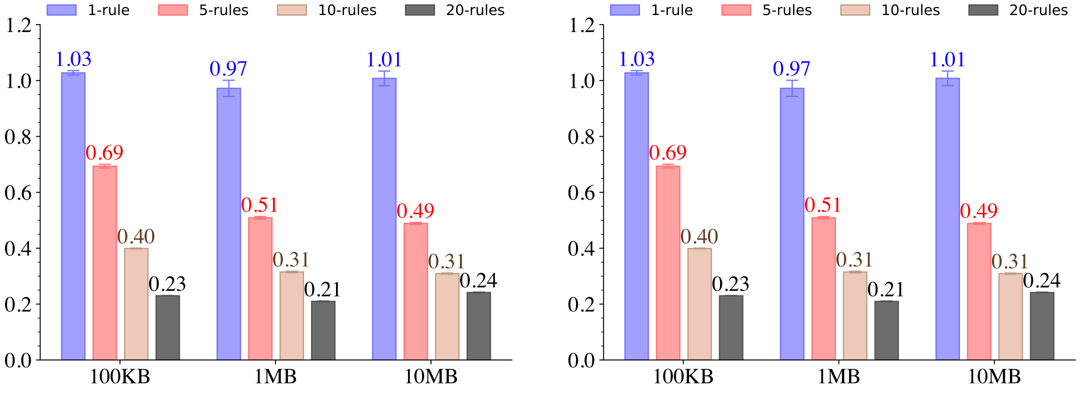
(a) Only `ipv6`

(b) `ipv6` and `nf_tables`

Figure 5-3: Average HAKC operations per second and per KB transmitted while running ApacheBench.

5.2.2 `nf_tables`

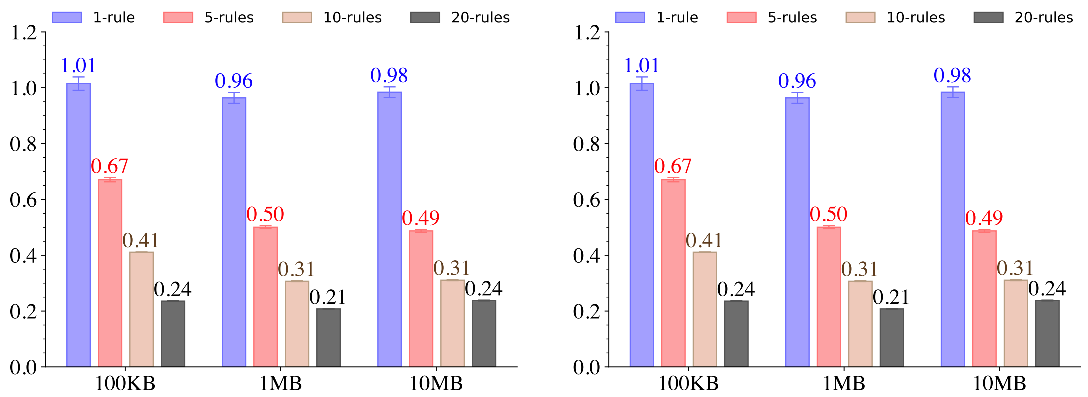
`nf_tables` was compartmentalized as a single clique, using a different compartment I.D. and clique color from those used for `ipv6` compartmentalization. As with `ipv6`, ApacheBench was also used to benchmark `nf_tables`, and all benchmark results are normalized against the unmodified kernel. Each benchmark measures performance for a single variable set to various parameters. Note that request and transfer results appear equivalent as a result of truncation of normalized overhead values.



(a) Requests/second

(b) Transfers(KB)/second

Figure 5-4: Increasing rules with hook: *INPUT*, rule: *DROP* source address



(a) Requests/second

(b) Transfers(KB)/second

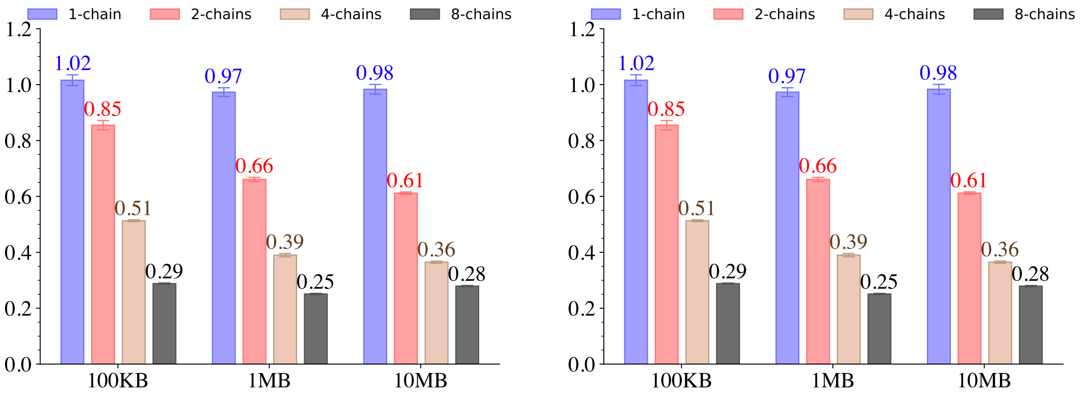
Figure 5-5: Increasing rules with hook: *INPUT*, rule: *ACCEPT* destination address

Figure 5 − 4 details the change in request per second and data transfer per second measurements as rules increase on the input hook. All rules were placed on a single chain. Notice the percentage difference in standard deviation of 5, 10, and 20 rule experimental results between 100KB and 10MB file transfer is around -44%. This suggests the increase in file size transfer incurs extraneous overhead that increasingly dominates the overhead incurred by packet filtering.

The experiment used to generate the measurements in Figure 5 − 5 only differs from the experiment associated with Figure 5 − 4 in the type of rule used to match packets. Rather than *accepting* packets for a matching IPv6 *destination* address, this experiment *drops* packets for a matching IPv6 *source* address. The results in Figure 5 − 5 are well within one standard deviation of the results in Figure 5 − 4. Therefore, we can conclude that there is no significant difference between performance of differing packet filtering rules.

Figure 5 − 6 demonstrates change in performance as the number of chains increase. Each experiment was run with one rule per chain. Notice the percentage difference in standard deviation of 5, 10, and 20 rule experimental results between 100KB and 10MB file transfers is around -40%. As with previously discussed experiments, this suggests the increase in file size transfer incurs extraneous overhead that increasingly dominates the overhead incurred by packet filtering.

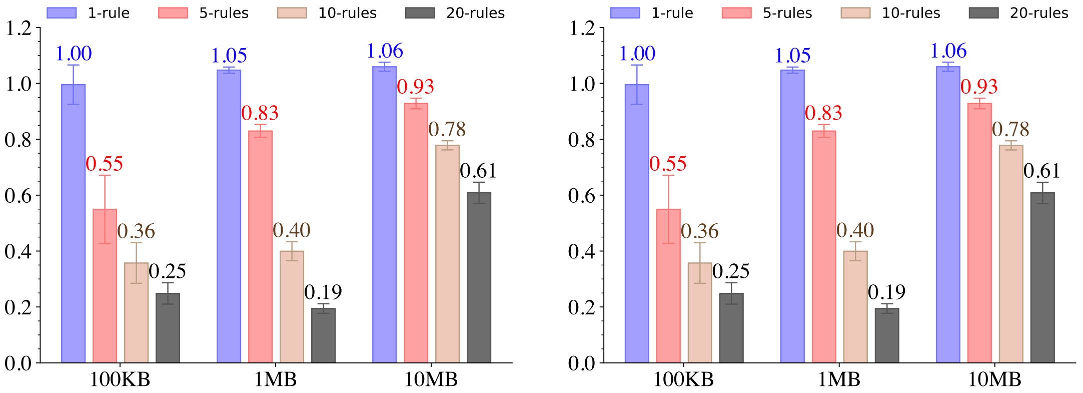
Figure 5 − 7 details experimental results of change in performance as rules increase



(a) Requests/second

(b) Transfers(KB)/second

Figure 5-6: Increasing chains with hook: *INPUT*, rule: *DROP* source address



(a) Requests/second

(b) Transfers(KB)/second

Figure 5-7: Increasing rules with hook: *OUTPUT*, rule: *ACCEPT* destination address

on the output hook. All rules were placed on a single chain. Unlike the input hook, which processes all incoming packets that are directed to a particular machine (excluding packets that are forwarded to other machines), the output hook processes all packets that leave a machine (again, excluding forwarded packets). The average percentage difference between 10MB and 100KB file transfers for 5, 10, and 20 rule experimental results is around -51%. This indicates that performance increases as file size increases. From the data presented, we can only infer the explanations of this trend. One possible explanation could be that overhead incurred by larger file sizes dominates packet filtering on the output chain. Another possibility is that op-

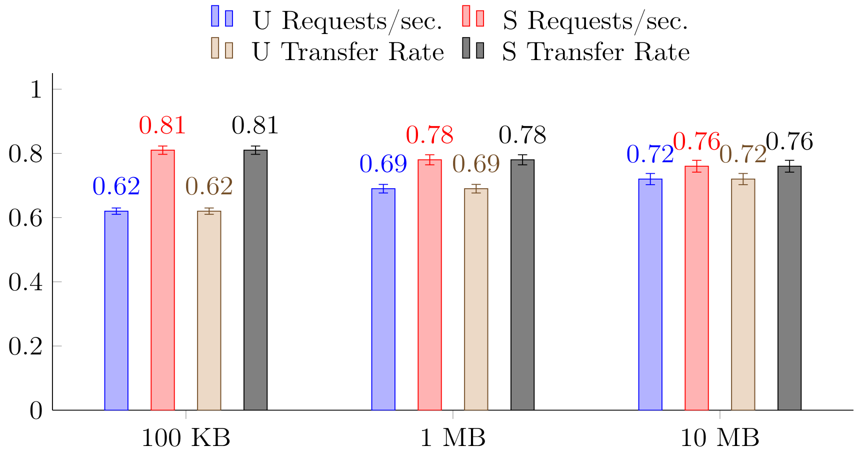


Figure 5-8: Overhead imposed when using multiple Compartments in a single system, normalized to the unmodified kernel (U) and single Compartments systems (S).

timizations are made for packets consisting of the same IPv6 data for a given set of rules.

Overall, there is around a 40% decrease in performance for the doubling of rules and chains. This is expected because rules and chains are applied in a linear fashion. This effect of compounding overhead may be partially related to the compartmentalization policy tested. We leave the exploration of change in performance incurred by different compartmentalization policies for future work.

5.3 Multiple Compartment System Overhead

Performance was also tested when both `ipv6` and `nf_tables` LKMs are compartmentalized. Each LKMs compartmentalization policies forbid direct mutual transitions.

To measure the overhead of using both LKMs on the same system, we defined a packet filter rule that drops packets with a source address from a specific IPv6 address. We then ran our microbenchmark detailed in Section 5.2 using the unmodified kernel, the compartmentalized kernel with only the `ipv6` LKM compartmentalized, and the compartmentalized kernel with both HAKC LKMs enabled. The results, normalized to the unmodified kernel (U) and `ipv6`-only (S) kernel overheads, are listed in Figure 5 − 8.

Website	Delta (s)	Stdev (s)
linkedin.com	-0.47	0.065
hdfcbank.com	-0.12	0.085
google.cn	-0.068	0.086
bing.com	-0.087	0.13
investing.com	38	62
okezone.com	-11	20
cnn.com	-9.8	15
yahoo.com	-4.9	15

Table 5.1: The measured time differences between the compartmentalized kernel of the lowest and highest standard deviations of unmodified kernel load times. Negative delta numbers indicate slower compartmentalized load time.

The general trend regarding payload size and overhead shown in Figure 5 − 2 is again present for the overhead against the unmodified kernel. However, the performance relative to the single compartment system degrades with payload size. The performance degradation comes from the additional compartment transitions the kernel makes to perform both packet filtering and TCP functionality with every TCP ACK packet received. This behavior is shown in Figure 5−3b, with the higher number of data pointer authentications per kilobyte than with just IPv6 compartmentalized. Regardless, Figure 5−8 shows a linear growth of 14%–19% per compartment *when the compartments are related, but provide orthogonal functionality*. Compartmentalizing both the IPv6 and packet filtering represents a worst case for performance loss, since all HAKC operations for both LKMs will occur in tandem, and will thus be directly compounded. A better compartmentalization policy will likely amortize individual overheads to a lower total overhead, but we leave that evaluation for future work.

5.4 User Website Browsing

Using ApacheBench to measure raw performance does not necessarily provide a good indication of whether a user will notice any performance difference when using the compartmentalized kernel for everyday activities. For example, activities unrelated to the kernel networking stack, such as routing delays, website rendering, or advertisement negotiation, can add significant time to end-user web page loading. To answer

Research Question 3, we want to measure any significant difference in IPv6 website loading time between using the unmodified kernel and our compartmentalized `ipv6` LKM given these external factors.

To that end, we created a Selenium script that spawns a headless Firefox instance, and proceeds to play a specific YouTube video, and then visits the 50 most popular websites (as determined by the Alexa Top 1M) that advertise an IPv6 address in their DNS Authoritative Record (an `AAAA` entry). We disable all memory and disk cache use and enable IPv6 use in Firefox. Additionally, before retrieving each website, we delete all cookies, and perform a DNS query to ensure that ISP DNS entries are fresh. Afterwords, we measure the time the Selenium web driver takes to fully render the page, or the time the YouTube video takes to complete. To account for possible differences in advertisements, we retrieve each website using the unmodified kernel and compartmentalized kernel in turn before retrieving the next website. We repeated this experiment 5 times, with each retrieval separated by approximately 1 hour.

Overall, we measured the average load time of the compartmentalized kernel to be 1.19 ± 4.34 seconds slower than the unmodified kernel. Because the standard deviation of load time differences is much larger than the average, we conclude that the compartmentalized kernel is not significantly different from the unmodified kernel, and that a user will not notice a difference using a compartmentalized kernel.

Despite our efforts to mitigate any possible difference between website retrievals, we did measure large differences in load times of some websites, on both large and short time retrieval spans. For example, `investing.com` would sometimes load in 4 seconds, and then after rebooting into a new kernel, the website would take 151 seconds. For this reason, we did not include `investing.com` in the average cited above. We were unable to determine any correlation between time of day or kernel type; the same website would be slow for the unmodified kernel at one time, and similarly slow for the compartmentalized kernel at another time, while the different kernels would statistically tie at every other time.

Table 5.1 lists the websites with the smallest and largest unmodified kernel load time standard deviations, along with the measured time differences when using HAKC.

```
int ip6_find_1stfragopt(struct sk_buff *skb, u8 **nexthdr)
{
    u16 offset = sizeof(struct ipv6hdr);
    struct ipv6_opt_hdr *exthdr =
          (struct ipv6_opt_hdr *)(ipv6_hdr(skb) + 1);
    unsigned int packet_len = skb_tail_pointer(skb) -
                              skb_network_header(skb);
    /* ... */
    while (offset + 1 <= packet_len) {
        struct ipv6_opt_hdr *exthdr;
        switch (**nexthdr) {
            /* ... */
        }
        offset += ipv6_optlen(exthdr);
        *nexthdr = &exthdr->nexthdr;
        exthdr = (struct ipv6_opt_hdr *)
             (skb_network_header(skb) + offset);
    }

    return offset;
}
```

Listing 5.1: CVE-2017-9074

In total, 20% (10/49) of the websites were measured to be faster using the compartmentalized kernel, and in all but one case, the load time delta was within 2 standard deviations (95% confidence). This provides further evidence that HAKC compartmentalization would go unnoticed by users in everyday usage.

5.5 Security Evaluation – CVE Case Studies

To provide a security evaluation on real-world bugs, we will examine two chosen vulnerabilities which will illustrate HAKC protection against bugs within and outside of compartmentalized code: CVE-2017-9074 [56] and CVE-2019-14815 [57]. CVE-2017-9074 is an internal IPv6 internal bug, while CVE-2019-14815 is an external bug in the Marvell Wifi driver. Of the 567 CVEs in our analysis set (see Chapter 1), only 12 involved IPv6, demonstrating the importance of having compartments be hardened against external bugs, as most kernel bugs will be outside of a compartment.

CVE-2017-9074 (Listing 5.1) allows for reading memory outside the bounds of the intended object. The bug involves a missing check on `offset` against `packet_len` that ensures that the code is reading within the bounds of the socket buffer, `skb`. Through a series of system calls, a malicious user can craft an IPv6 packet that

```
void mwifiex_set_uap_rates(
     struct mwifiex_uap_bss_param *bss_cfg,
     struct cfg80211_ap_settings *params) {
    struct ieee_types_header *rate_ie;
    /* ... */

    rate_ie = (void *)cfg80211_find_ie(WLAN_EID_SUPP_RATES, var_pos, len);
    if (rate_ie) {
        memcpy(bss_cfg->rates, rate_ie + 1,
               rate_ie->len);
        rate_len = rate_ie->len;
    }

    rate_ie = (void *)cfg80211_find_ie(
            WLAN_EID_EXT_SUPP_RATES,
            params->beacon.tail,
            params->beacon.tail_len);
    if (rate_ie)
        memcpy(bss_cfg->rates + rate_len, rate_ie + 1,
               rate_ie->len);

    return;
}
```

Listing 5.2: CVE-2019-14815

contains an invalid option, which causes `offset` to be much larger than the size of the allocated buffer for `skb`. `offset` is used to compute `*nexthdr`, which is read in the switch statement. This read is the out-of-bounds memory read.

HAKC prevents arbitrary out-of-bounds memory accesses like this, and instead limits the code's ability to only access the data explicitly allowable by the clique `ip6_find_1stfragopt` belongs to. The large, corrupted `offset` value can place `exthdr` in one of several places: 1) a different compartment and a different colored clique; 2) a different compartment but the same colored clique; 3) the same compartment and a different colored clique; and 4) the same compartment and the same clique. In the first two situations, PAC authentication will fail because the computed PAC context will not match the PAC context used to sign `exthdr`. The third situation allows access only if the clique is accessible according to the defined access-control policy, and the fourth situation will be allowed by HAKC.

To successfully perform this out-of-bounds read on HAKC-protected code, the attacker would have to construct `offset` such that the resultant pointer points to an accessible clique, *and* contains the correct signature. The first condition already limits arbitrary accesses, and the second condition is computationally hard. This is how HAKC compartmentalizes code and data. The attacker is able to only access data allowed by the access-control policy, even in the presence of bugs, *and* the attacker must perform a computationally hard task to do so.

CVE-2019-14815 (Listing 5.2) is a bug in the Marvell Wifi driver that uses data from user-space in `memcpy` without checking the data length, leading to a heap overflow. Assume that the attacker uses this CVE from uncompartmentalized code to overwrite a pointer in compartmentalized code. The new pointer must again conform to all data access policies, and must contain a valid signature for the new pointer. Only if the new pointer is validly accessible and correctly signed, then the attack will succeed. However, as mentioned earlier, satisfying all the conditions is computationally hard.

Unfortunately, non-pointer compartmentalized data can be corrupted. However, this will likely only cause a denial of service, which, though severe, is considered less serious than privilege escalation. One mitigation would be to utilize the "traditional" MTE, and store the color in the pointer along with the PAC signature. The MTE hardware can check the color of accessed addresses, and check that value with stored value, and throw a fault if they mismatch. The use of MTE and PAC in this way reduces the available signature bits by half, making brute force guessing of a signature easier.

Chapter 6

Discussion

The following sections outline future plans for HAKC regarding improving ease-of-use, performance, and breadth of application.

6.1 Performance Improvement

Although intra-procedural and inter-procedural LLVM analysis significantly reduce unnecessary signing and authentication overhead, there may still exist opportunities for performance improvement. One strategy that could possibly yield performance gains is checking for repeated pointer authentication across *phi node* frontiers.

In LLVM, phi nodes represent static single-assignment (SSA) registers whose values come from a set of other registers but cannot be determined until runtime. This is a common situation that is the result of SSA call graphs. For example, consider the variable that is defined outside of an if block, and then reassigned inside that if block. In LLVM, this variable would be represented by a phi node, with two parent registers. The first parent is in one basic block and the second is in another basic block. Currently, pointer sources are ascertained by traversing backwards through LLVM code, stopping at phi nodes. However, there is an opportunity for improvement here. Rather than stopping the source search at a phi node, we can check if the registers of all phi node input registers are authenticated. This will be a recursive check, stopping when either we have reached a maximal depth, encountered the source of the pointer, or have encountered an authentication. This strategy of recursively checking phi node input registers can further reduce unnecessary checks.

Leaf functions are functions in a compilation unit that do not make calls to other functions. Leaf functions can be called from within or outside a translation unit. Leaf functions may present an opportunity for performance improvement by inserting copies of a leaf function into the translation unit that calls a leaf function. If the leaf function is only called within a given compartment/clique, then the leaf functions arguments may not need to be authenticated. This comes with a trade-off however, because injecting leaf functions at there call sites will increases the size of the final kernel image.

6.2 Automation

There are two facets to automation that are worth highlighting. The first component is decreased developer instrumentation, and the second is automatic compartmentalization schemes that allow for security/performance trade-off testing. One way to improve automation in either of these areas is by providing a graphical tool for developers to easily select files and functions that belong in defined compartments/-cliques. This would provide the LLVM pass with knowledge of what functions are not part of a compartmentalization scheme, allowing for increased transfer instrumentation automation. Secondly, various compartmentalization tools could then be used to generate compartmentalization policies that can be tested by the developer.

6.3 Other Areas of Application

Although HAKC has only been tested for LKM compartmentalization, HAKC can be applied to other parts of the kernel as well as user space applications. For example, HAKC can be used to compartmentalize unsafe driver code and extended Berkely Packet Filtering (eBPF). eBPF is user space injected code into the kernel that can be attached to hooks and make calls to certain modules. Although eBPF is sanitized before execution and is not turing complete, the Kernel could benefit from running eBPF programs in compartmentalized memory spaces, further bolstering there security. Regarding user space applications, HAKC could be used to secure real-time systems such as ROS, a middleware typically used for inter-process communication in autonomous systems (such as satellites and robots). Since HAKC relies on hardware to enforce compartmentalization, HAKC could assist efforts in compartmentalizing hypervisors [58].

Chapter 7

Related Work

Efforts have been made to devise hardware based strategies to enforce compartmentalization schemes to avoid software overhead. The following sections compare HAKC to various hardware enforced compartmentalization efforts that are intended for user or kernel space.

7.1 Hardware Safe Region Enforcement

Other works have focused on using hardware to support *safe regions* — regions of memory only accessible by privileged instructions — but have only extended simulated hardware and have focused on user-space applications [59, 60]. One such example is Microstache [59], which allocates a small region of memory that can only be accessed by privileged instructions. To protect against side-channel attacks, safe region memory is cached in a separate designated cache to mitigate exposure to side-channel attacks. Imix [60] is an alternative safe region solution for x86 architectures, allow pages to be tagged with a *security sensitive* permission that can only be accessed by privileged instructions. IMIX is intended to be used as a security primitive, providing support for more complex security schemes that support control-flow and code-pointer integrity (CPI).

7.2 Hardware Pointer Tagging

One way to spatial memory safety is to associate bounds with each pointer that can be checked at runtime. Traditionally, software based bounds checking schemes incurs significant overhead. However, several efforts have focused on using hardware to enforce pointer bounds checking. One such example is One solution, Hardbound [61], allows software to define pointer bounds during runtime. These bounds are then enforced by hardware during pointer loads and stores. Another approach, In-Fat Pointer [62], which is able to achieve intra-object bounds granularity by storing metadata in the top 16 unused bits of pointers and storing tables that track sub-object types. Unfortunately, neither of these solutions were implemented in conventional hardware, and only In-Fat Pointer comes close to provide comparable granularity to that of Arm MTE.

7.2.1 Intel MPK

Recent work has focused on extending hardware to support page-granularity protection keys. One such example is Intel Memory Protection Keys (MPK). MPK is an x86 extension that allows individual pages to be tagged with a 4-bit domain id. This allows a processes's address space to be partitioned into at most 16 domains. Each logical core also has a special register, PKRU, that dictates per-domain read and write protections. The PKRU can be written to from user space by different threads within a process, allowing for more flexibly and faster permission changes compared to a traditional system call that modifies PTE flags.

MPK's lack of restrictions on a processes's ability to set page keys and the PKRU increases the attack surface of a process if MPK related system calls and instructions are used irresponsibly. An attacker can easily change values in the PKRU to grant access to all memory regions if the process is compromised. To combat this issue, solutions have been proposed that use MPK to enforce isolation in a safe manner [63, 64, 65, 66]. It should be noted that some of these solutions shown to be vulnerable to various attacks that leverage Linux kernel system calls to subvert MPK permissions [67].

Another key based hardware solution, Donky [68], utilizes the top 10 bits of PTEs on x86 and RISC-V architectures to store keys, allowing for 1024 possible memory domains. Donky also provides a special user space protection key policy register on x86 architectures that requires access through a call gate to a trusted handler, alleviating the need for binary inspection of MPK-based applications to ensure safe register use.

7.2.2 Intel SGX

Intel Software Guard Extensions (SGX) is an x86 hardware extensions that enables both users and the operating system to define cryptographically protected enclaves. All computation within these enclaves is trusted, but everything outside of an enclave, including the operating system and hypervisor. Research efforts have shown that SGX is vulnerable to side-channel attacks and is unable to protect malicious code running inside an enclave from accessing data outside the enclave [69, 70].

7.2.3 Hardware Memory Tagging and Policy Enforcement

There are several works on using hardware tagging to support various compartmentalization and pointer bounds checking schemes [71, 72, 73], however most of these works are implemented on simulated architectures. One effort in particular, Mondrix [74], provides inter-modular Linux kernel compartmentalization using a 2-bit word granularity tagging extension [75]. Unlike HAKC, Mondrix is implemented in simulation, and requires a memory supervisor that monitors all kernel permission changes. Furthermore, Mondrix only implements inter-module isolation, whereas HAKC supports both inter-module and intra-module isolation.

Dover [76], an SoC that allows developers to tag each word in memory with metadata, and use that metadata to enforce various defined policies upon execution of each instruction. Dover designates a separate execution core for enforcing policies in parallel during program execution. Another such architecture is CHERI [77], which is intended to provide hardware capability enforcement by minimally extending current existing architectures such as RISC-V and ArmV8-A. This architecture works by storing double word length capabilities that contain four capability elements that each help enforce a protection model for the associated memory address. The architecture also requires the maintaining of 1 bit validity tags for each capability. Arm recently released Morello, an ArmV8-A architecture profile that implements a variant of CHERI [77]. CheriBSD is a recompiled and slightly modified version of FreeBSD that has been used to test the performance impacts of the CHERI extension. Benchmarks have shown little performance impact on operating system code, but somewhat significant performance impact on applications like language runtimes [77].Cheri's fixed capability model provides less flexibility than PAC, where arbitrary information can be used as the context to sign pointers. Further, Cheri's focus on capabilities misses data-only attacks.

7.3 Arm PAC/MTE Efforts

Recent works have utilized Arm PAC to enforce control-flow integrity (CFI), spatial memory safety, and code pointer integrity (CPI). PACStack [78] is a CFI scheme that secures return addresses stored on the stack through a chain of hashing, where a hash for each return pointer is unique based on the current execution path of a program. PTAuth [79] enforces temporal memory safety by storing a unique id at the base of data object, using the unique id as the PAC context during signing and authentication. PARTS [54] is an LLVM instrumentation framework that utilizes PAC to support a CPI scheme that is resistant to pointer-reuse attacks, and thwarts control-flow and data-oriented attacks. Compared to these schemes, HAKC can provide wider protection against many classes of attacks, and in some cases, like with PACStack, can be used in conjunction. HAKC is the first design to the best of our knowledge that utilize MTE-based isolation. However, designs have been proposed that would leverage MTE-like architectural features to improve the Clang AddressSanitizer [80].

Chapter 8

Conclusion

With the prevalence of monolithic kernel architectures and their relevance to the growth of IoT, the issue of enforcing least privilege remains a very serious and relevant concern whose potential solutions demand further research. Leveraging Arm PAC and MTE, HAKC enforces least privileges by supporting a performant solution to compartmentalization that, unlike past compartmentalization efforts, can be implemented using commodity hardware and LLVM instrumentation. Using a two-tier compartmentalization scheme that prioritizes inter-clique performance, developers are afforded substantial flexibility in the number of compartments at their disposal and are able to easily modify compartmentalization policies to test security-performance trade-offs. Though HAKC is tested for LKM use, a lack of dependence on a TCB makes HAKC an avenue of interest for compartmentalization outside the kernel. Web servers, which hold sensitive user information largely exist as monoliths, and hypervisors, which struggle with security vulnerabilities incurred by monolithic design, are two examples of systems that should be prioritized highly as future research for HAKC compartmentalization.

Bibliography

[1] J. Bradley Chen and Brian N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, page 120–133, New York, NY, USA, 1993. Association for Computing Machinery. URL: https://doi-org.libproxy.mit.edu/10.1145/168619.168629, doi:10.1145/168619.168629.

[2] Turbo Future. Home applications of linux-based operating systems, 2021. URL: https://turbofuture.com/computers/Home-Applications-of-Linux-Based-Operating-Systems.

[3] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[4] The kernel development community. The kernel address sanitizer (kasan), 2021. URL: https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html.

[5] Intel. Netfilter how-to, 2018. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/best-practices/related-intel-security-features-technologies.html.

[6] LTD. ARM. Pointer authentication on armv8.3, 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[7] LTD. ARM. Armv8.5-a memory tagging extension, 2019. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[8] Intel. Intel® 64 and ia-32 architectures software developer's. URL: https://software.intel.com/content/www/us/en/develop/download/cfiapplicatoinicintel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c html.

[9] LTD. ARM. Armv8-m architecture reference manual. URL: https://documentation-service.arm.com/static/60e6f8573d73a34b640e0cee.

[10] Martín Abadi, Úlfar Budiu, Mihaiand Erlingsson, and Jay Ligatti. Control-flow integrityprinciples, implementations, and applications. *ACMTrans. Inf. Syst. Secur.*, 13(1), November 2009. doi:10.1145/1609956.1609960.

[11] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999, 2019. doi:10.1109/SP.2019.00076.

[12] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019. doi:10.1109/SP.2019.00002.

[13] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[14] R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205, 2013. doi:10.1109/SP.2013.23.

[15] Shuo Chen, Jun Xu, and Emre C. Sezer. Non-control-data attacks are realistic threats. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD, July 2005. USENIX Association. URL: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[16] NIST. Cve-2016-4997 detail, 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-4997.

[17] Justin Restivo. A zero kernel operating system: Rethinking microkernel design by leveraging tagged architectures and memory-safe languages. Master's thesis, Massachusetts Institute of Technology, United States, 2019.

[18] MITRE Corp. Cve details, 2019. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[19] Minix 3. https://wiki.minix3.org/doku.php?id=www:documentation:reliability. Accessed: 2020-12-16.

[20] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. doi:10.1145/1243418.1243424.

[21] Jorrit Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew Tanenbaum. Construction of a highly dependable operating system. In *2006 Sixth European Dependable Computing Conference*, pages 3 – 12, 12 2006. doi:10.1109/EDCC.2006.7.

[22] RedoxOS. https://redox-os.org. Accessed: 2020-12-16.

[23] Rust. The rust programming language. URL: https://doc.rust-lang.org/book/.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the*

Fifteenth ACM Symposium on Operating Systems Principles, SOSP '95, page 251–266, New York, NY, USA, 1995. Association for Computing Machinery. doi:10.1145/224056.224076.

[25] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. *Unikraft: Fast, Specialized Unikernels the Easy Way*, page 376–394. Association for Computing Machinery, New York, NY, USA, 2021. URL: https://doi.org/10.1145/3447786.3456248.

[26] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, May 2015. USENIX Association. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy.

[27] Maxime Compastié, Rémi Badonnel, Olivier Festor, Ruan He, and Mohamed Kassi-Lahlou. Unikernel-based approach for software-defined security in cloud infrastructures. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018. doi:10.1109/NOMS.2018.8406155.

[28] Anil Madhavapeddy and David J. Scott. Unikernels: Rise of the virtual library operating system: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework? *Queue*, 11(11):30–44, December 2013. doi:10.1145/2557963.2566628.

[29] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. Fades: Fine-grained edge offloading with unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, HotConNet '17, page 36–41, New

York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3094405.3094412.

[30] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, page 8–15, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3365137.3365395.

[31] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 143–156, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381326.

[32] Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the Twenty-Fourth ACMSymposium on Operating Systems Principles*, SOSP '13, page 116–132, New York, NY, USA, 2013. Association forComputing Machinery. doi:10.1145/2517349.2522719.

[33] Keir Fraser, Steven H, Rolf Neugebauer, Ian Pratt, and Mark Williamson. Safe hardware access with the xen virtual machine monitor. In *In Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, 2004.

[34] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3381052.3381328.

[35] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. Lxds: Towards isolation of kernel

subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association. URL: `https://www.usenix.org/conference/atc19/presentation/narayanan`.

[36] 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 *In Network and Distributed System Security Symposium (NDSS)*, 2019. `doi:10.14722/ndss.2019.23068`.

[37] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 65–82, 2018.

[38] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security (NDSS) Symposium*, February 2018. `doi:10.14722/ndss.2018.23107`.

[39] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015. `doi:10.1145/2744769.2744847`.

[40] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 443–446, 2008. `doi:10.1109/ASE.2008.69`.

[41] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automaticgeneration of high-coverage tests for complex systems programs. In *Proceedings of the 8thUSENIX Conference on Operating Systems Design andImplementation*, OSDI'08, page 209–224, USA, 2008. USENIX Association.

[42] janala2. Catg, 2012. URL: `https://github.com/ksen007/janala2`.

[43] Golang. https://golang.org. Accessed: 2020-7-25.

[44] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013. doi:10.1109/SP.2013.13.

[45] Vytautas Astrauskas, Federico Matheja, Christophand Poli, Peter Müller, and Alexander J. Summers. How do programmers use unsafe rust? *Proc. ACMProgram. Lang.*, 4(OOPSLA), November 2020. doi:10.1145/3428204.

[46] Cody Cutler, M. Frans Kaashoek, and Robert T. M orris. 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, Carlsbad, CA, October 2018. USENIX Association. URL: https://www.usenix.org/conference/osdi18/presentation/cutler.

[47] TockOS. https://www.tockos.org. Accessed: 2020-12-16.

[48] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007. doi:10.1145/1243418.1243424.

[49] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in linux. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010. URL: https://www.usenix.org/conference/usenix-atc-10/tolerating-malicious-device-drivers-linux.

[50] Roberto Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, 2017(1):4–44, Mar. 2017. URL: https://tosc.iacr.org/index.php/ToSC/article/view/583, doi:10.13154/tosc.v2017.i1.4-44.

[51] Nokia Corporation. The linux networking stack architecture, 2001. URL: `http://affix.sourceforge.net/affix-doc/c190.html`.

[52] paulgorman.org. Netfilter, 2018. URL: `https://paulgorman.org/technical/linux-nftables.txt.html`.

[53] Rusty Russel and Harald Welte. Netfilter how-to, 2002. URL: `https://www.netfilter.org/documentation/HOWTO/netfilter-hacking-HOWTO.txt`.

[54] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 177–194, USA, 2019. USENIX Association.

[55] The Apache Software Foundation. ab - apache http server benchmarking tool. URL: `http://httpd.apache.org/docs/2.4/programs/ab.html`.

[56] NIST. Cve-2017-9074 detail, 2021. URL: `https://nvd.nist.gov/vuln/detail/CVE-2017-9074`.

[57] NIST. Cve-2019-14815 detail, 2021. URL: `https://nvd.nist.gov/vuln/detail/CVE-2019-14815`.

[58] Lei Shi, Yuming Wu, Yubin Xia, Nathan Dautenhahn, Haibo Chen, Binyu Zang, and Jinming Li. Deconstructing xen. In *NDSS*, 2017.

[59] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. Microstache: A lightweight execution context for in-process safe region isolation. In Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis, editors, *Research in Attacks, Intrusions, and Defenses*, pages 359–379, Cham, 2018. Springer International Publishing.

[60] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, Baltimore, MD, Au-

gust 2018. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/frassetto.

[61] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. *SIGOPS Oper. Syst. Rev.*, 42(2):103–114, March 2008. URL: https://doi-org.libproxy.mit.edu/10.1145/1353535.1346295, doi:10.1145/1353535.1346295.

[62] Shengjie Xu, Wei Huang, and David Lie. In-fat pointer: Hardware-assisted tagged-pointer spatial memory safety defense with subobject granularity protection. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 224–240, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3445814.3446761.

[63] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[64] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/hedayati-hodor.

[65] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *2019*

USENIX Annual Technical Conference (USENIX ATC 19), pages 241–254, Renton, WA, July 2019. USENIX Association. URL: `https://www.usenix.org/conference/atc19/presentation/park-soyeon`.

[66] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating function-as-a-service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021. URL: `https://www.usenix.org/conference/atc21/presentation/kotni`.

[67] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on pku-based memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426. USENIX Association, August 2020. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/connor`.

[68] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel`.

[69] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks, 2019. `arXiv:1702.08719`.

[70] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel sgx, 2019. `arXiv:1902.03256`.

[71] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted dataflow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17, 2016. `doi:10.1109/SP.2016.9`.

[72] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3064176.3064217.

[73] 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, 2018. doi:10.1109/SP.2018.00066.

[74] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 31–44, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1095810.1095814.

[75] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 304–316, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/605397.605429.

[76] A. A. d. Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. Micro-policies: Formally verified, tag-based security monitors. In *2015 IEEE Symposium on Security and Privacy*, pages 813–830, 2015. doi:10.1109/SP.2015.55.

[77] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-941.pdf.

[78] Hans Liljestrand, Thomas Nyman, Lachlan J. Gunn, Jan-Erik Ekberg, and N. Asokan. Pacstack: an authenticated call stack. In *30th USENIX*

Security Symposium (USENIX Security 21). USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[79] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. Ptauth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., August 2021. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/mirzazade.

[80] Clang. Hardware-assisted addresssanitizer design documentation. URL: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[81] Microsoft patches windows kernel flaw under active attack. https://www.darkreading.com/threat-intelligence/microsoft-patches-windows-kernel-flaw-under-active-attack/d/d-id/1339415. Accessed: 2020-12-16.

[82] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451167.

[83] Ferdinand Brasser, David Gens, Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. Sanctuary: Arming trustzone with user-space enclaves. In *Network and Distributed Systems Security (NDSS) Symposium*, February 2019. doi:10.14722/ndss.2019.23448.

[84] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery. URL: https://doi-org.libproxy.mit.edu/10.1145/3427228.3427262, doi:10.1145/3427228.3427262.

[85] Dawei Chu, Yuewu Wang, Lingguang Lei, Yanchu Li, Jiwu Jing, and Kun Sun. Ocram-assisted sensitive data protection on arm-based platform. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security – ESORICS 2019*, pages 412–438, Cham, 2019. Springer International Publishing.

[86] Shengye Wan, Mingshen Sun, Kun Sun, Ning Zhang, and Xu He. Rustee: Developing memory-safe arm trustzone applications. In *Annual Computer Security Applications Conference*, ACSAC '20, page 442–453, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3427228.3427262.

[87] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the linux kernel. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 315–332, Cham, 2019. Springer International Publishing.

[88] ARM. Trustzone. URL: https://developer.arm.com/ip-products/security-ip/trustzone.

[89] Intel. Intel® software guard extensions. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html.