PROTECTION IN COMMODITY MONOLITHIC OPERATING SYSTEMS

BY

NATHAN DANIEL DAUTENHAHN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Doctoral Committee:

Professor Vikram Adve, Chair and Director of Research
Professor Klara Nahrstedt
Professor Carl A. Gunter
Adjunct Professor Edouard Bugnion, EPFL
Dr. Samuel T. King, Twitter

Abstract

This dissertation suggests and partially demonstrates that it is feasible to retrofit real privilege separation within commodity operating systems by “nesting” a small memory management protection domain inside a monolithic kernel’s single-address space: all the while allowing both domains to operate at the same hardware privilege level. This dissertation also demonstrates a microarchitectural return-integrity protection domain that efficiently asserts dynamic “return-to-sender” semantics for all operating system return control-flow operations. Employing these protection domains, we provide mitigations to large classes of kernel attacks such as code injection and return-oriented programming and deploy information protection policies that are not feasible with existing systems.

Operating systems form the foundation of information protection in multiprogramming environments. Unfortunately, today’s commodity operating systems employ monolithic kernel design, where any single exploit in the vast code base undermines all information protection in the system because all kernel code operates with full supervisor privileges, meaning that even perfectly secure applications are vulnerable.

This dissertation explores an approach that retrofits fundamental information protection design principles into commodity monolithic operating systems, the aim of which is a micro-evolution of commodity system design that incrementally decomposes monolithic operating systems from the ground up, thereby applying microkernel-like security properties for billions of users worldwide. The key contribution is the creation of a new operating system organization, the Nested Kernel Architecture, which "nests" a new, efficient intra-kernel memory isolation mechanism into a traditional monolithic operating system design. Using the Nested Kernel Architecture we introduce write-protection services for kernel developers to deploy security policies in ways not possible in current systems—while greatly reducing the trusted computing base—and demonstrate the value of these services by deploying three special data protection policies.

Overall, the Nested Kernel Architecture demonstrates practical in-place protections that require only minor code modifications with minimal runtime overheads.

Dedicated to the love of my life, Audrey Dautenhahn, and those three lovely boys who have joined us along the way: Noah, Levi, and Zeke.

Acknowledgments

Pursuing a doctorate has been the biggest challenge I have ever pursued in my life—immeasurably and indescribably so. It has forced me to peer into the deepest parts of my life and gain order there first, ere I ever set foot being productive toward the doctorate. I learned that, at least for me, progress depended upon maturity in heart and mind. Thus my first acknowledgment and show of gratitude is towards my heavenly Father, the Spirit, and Son Jesus. For without your healing, guidance, direction, peace, fortitude, and rest, I would have quit long ago.

Right in the middle of this chaos has been the love of my life, Audrey, who just like me, has matched me step for step these seven years. Not only did we independently mature, but we also grew more entwined together. Your love, unwavering support, and awesome food engineering filled this student with hope, strength, encouragement, and resolve. Thank you My Lobster.

To my sons Noah, Levi, and Zeke, thank you so much for sharing some daddy time to get this doctorate completed: our adventures in Minecraft, wrestling, playing music, and unending hours of nighttime snuggles have been one of my greatest delights. I also want to commend you for listening to the many presentations I practiced under your watchful eyes during breakfasts. Know how greatly I love and appreciate you. Noah, I thank you for teaching me one of the most impacting lessons I have ever received in my graduate career: once I asked you how you made such great Lego creations, and you answered, "Well, Dad. I just start building until it looks like something I like. And then I keep building it." Indeed. Thank you.

Along the doctorate route I ran into a mirror: thank you Mark for helping the light not only shine into the deepest parts of me, but to also make sure I looked long enough to see what was there.

To my advisor, Vikram Adve, who took a chance on this late stage student. Your affinity to the most minute details and deep technical expertise—in pretty much everything I attempted—refined my thinking, designing, and building of secure systems. Although we share an affinity to optimism you always knew how to bring my optimism into practically attainable goals. You displayed tremendous trust in my abilities as a researcher, which enabled me to explore with confidence regardless of how I really felt. Thank you.

To John Criswell, thank you for opening up my eyes to the world of malicious operating systems and the vast complexity and fun of compiler based virtual machines, and LLVM! Thank you for your patience and gracefully teaching me the art of deep-SEE-diving.

To Theo and Will: it was an amazing adventure working with you on the Nested Kernel. Your dedication and trust to just go and do it was stupendous. Theo, you went above and beyond what I ever could expect, to design and implement crazy stuff in basically no time (a buddy allocator in a day, poster, system call auditing system, and on): if I didn't know better I'd say this wasn't your first rodeo, but it was. There is no doubt that my accomplishments come at great and direct impact from your efforts. Theo, you also provided feedback on so many elements of my various ideas and projects, always representing a sound and logically grounded position. Thank you. Will, you were a beast on PerspicuOS, and surprised me so many times with your ninja coding skillz. I also appreciate you joining with me in my quest to make graduate education better, if for nothing else, we both know a bit better who we are, what we stand for, and our aim: thanks.

To the LLVM and King research groups: you have truly aided me in my pursuit. Thank you for all the countless hours you have spent listening to presentations and reading various sections of papers and being available for a quick sanity check. Prakalp, you, being in the seat closest to mine while also competing for *earliest in*, have always been available for a quick review, chat, or to just here me vent. Maria, thank you for always listening to my rants and providing timely and although rough to hear reviews, always led to quality improvement. Hui, thank you for your sharing with me your insights on system building and how to truly mature as a systems researcher. Thanks to you all.

To Sam King, thank you for pursuing me and always encouraging me to go build something. You taught me that the best manual is always the code, and taking that to heart I was able to become much more efficient at systems development.

To Ed Bugnion, who joined me recently and sparked ideas for deconstructing monolithic operating systems automatically. Thank you Ed, your keen insight and questions have taught me much about systems design and implementation, and your suggestion that my doctorate is only the beginning of my career started a maturation in the way I view my time and goals as I transition from student to teacher.

To Michael Loui, who provided a venue for me to explore my interests in doctoral education. Who also taught me one of the most valuable and always relevant questions: how do you know? Thank you for your investment and joining me in my pursuit of an improved graduate education.

During the course of my graduate education I found that some of the most effective teachers were those who were only a few years down the road from me. To Nima Honarmand, mis amigo and co-conspirator in deterministically replaying the world, thank you for collaborating with me and showing me a level of grace and patience in academia I didn't think was possible. You also taught me much about how to find good research problems, be persistent in hard work, and navigate the complexities of obtaining a doctorate, thank you. To Matt Hicks, who I have had the pleasure of both working in the same research group with and collaborating on exciting SecRets, thank you for taking the time to deliberately invest in me and for providing guidance on too many occasions to count. You have exhibited a strong commitment to and awareness of my unique maturity and provided timely advice and encouragement: there was also that one time that even a body slam couldn't pry the ball out of my hands, we make a good team. Thank you.

To my thesis committee, thank you for your time and effort in providing this final review of my accomplishments in the doctorate program. A special thanks to Carl Gunter, who, although our paths only crossed ever so briefly, provided timely words of encouragement that provided a substrate of confidence and peace.

To David and Karen, thank you for your deep friendship and camaraderie throughout the PhD process. You were always there to listen to a rant and were awesome in making sure my arguments were logically sound. David, to completion of the Twilight Struggle and continued enjoyment of research after doctorate.

During my experiences as a doctorate I was fortunate to cross paths with truly exquisite researchers and members of the community. Thank you to Robert Watson, who helped me give my first talk outside of UIUC, and taught me about the social expectations of such an endeavor: this experience greatly enhanced my ability to go beyond the boundaries of my little doctorate world at home. Thank you to Gernot Heiser who talked shop with me about microkernels and gave me confidence that the Nested Kernel is an exciting and new direction of research. Thank you to JMS who listened to my rants about *doctoral education* being just that, an education, for revealing my value as a researcher, and setting a standard of mentoring and advising that I hope to one day match.

To the faculty at UIUC that provided a rich and detailed set of experiences to aid me in my quest to be a scholar, thank you. A special thanks to Roy Campbell, who in a single presentation alleviated so many of my doctorate fears by convincing me that I don't need to solve the world's biggest problems, just show that I can do research. Thanks.

To the scientific community that has refereed and reviewed several of my paper submissions over the years. One of the most amazing aspects of our communities is peer review, which has provided me insight in numerous projects I would not have had otherwise. Special thanks to the reviewers on the Nested Kernel and our shepherd, Peter Druschel, who all concisely and unambiguously pinpointed the contributions despite a somewhat convoluted introduction. I learned a tremendous amount from all of the anonymous reviewers during my doctorate, thank you.

To the faculty at the University of New Mexico for establishing my foundation and giving me an environment to explore exactly who I was to become. In particular, my passion for education was kindled at a very early time by Professor Salazar, who always answered my numerous questions with pinpoint precision and complete coverage to my eager satisfaction. Your teaching set the standard that I measure all teachers by and one that I hope to achieve in my career. To Greg Heileman, who allowed me to hack as an undergraduate researcher, thank you for your time and support throughout the process. To Nasir Ghani, who helped me transition from undergraduate researcher to graduate student, thank you for your support and encouragement in keeping focused on consistently working towards completion.

To Mr. Pierce, my high school English teacher, who went beyond merely teaching English. That course forced us to abandon our trivial high school world of phony-ness and cut through to deep heart issues, a rarity amidst a world where everyone was just trying to fit in. You, as a teacher, were not satisfied with telling us how to express ourselves in various written pros, but demonstrated it first hand to our great benefit. Thank you for providing a frank, honest, and truthful environment and for sparking in me a desire for education and its power to enhance quality of life.

To all of the amazing administrative support staff that made my life not miserable, thank you! Administration is messy, and something I'm not well suited for, to your efforts in helping me organize my courses, schedule conference rooms, handling reimbursements, purchasing equipment, and just all around awesomeness I'm extremely grateful.

To Mom and Gary and Dad and Denise, well its in the jeans (a.k.a. genes). Echoes of your encouragement, belief, and passion for me to reach my pinnacle are all over this dissertation. Thank you for your continued support and unconditional love. To Jordan, thank you for your support and the aid in several moves over the years. To Rachel, thank you for support in travel and cheerleading me along the way: you have always been my cheerleader.

Along the route to a doctorate I have had three lovely children, which is a lot of work. I would like to thank the community of family support that Audrey and I have obtained during the doctorate because, quite frankly, without it we would have been lost. To the Norcross family, you have truly blessed and uplifted our family through our relationships and set a standard for family living in the modern age. Additionally, your support in watching children throughout the entire process allowed us to focus on birthing siblings while knowing the others were safe: we are truly grateful for your friendship and thank you so much. Also, to our *date-night* group, thank you; how did we ever live without that?

Pursuing a doctorate is stressful, and without exercise and competition in sports it wouldn't be possible. Thank you to my racquetball stalwarts who ensured I ran all over the place, Patrick and David. Also thanks to the Sam King Basketball club: that was awesome, too bad we never saw Sam dunk.

A special thanks to my financial sponsors from the ONR via grant number N00014-12-1-0552.

I would like to thank last those researchers who have gone before me, who paved the way for all of my investigations and findings. We truly stand on the shoulders of giants as we wade into scholarly work. The particularly influential forerunners include: J. Saltzer, M. Schroeder, D. Parnas, D. Siewiorek, B. Lampson, E. Dijkstra, and the many many many more who contributed to my little slice of interests.

Table of Contents

Chapter 1

Introduction

Commodity, monolithic operating systems are trusted by billions, of users worldwide to perform safety and security critical operations. Not only are operating systems implicitly trusted by users to accomplish tasks such as banking, email, and taxes, they are also trusted for highly sensitive operations such as vehicle control (Mashable, 2015; Linux Foundation, 2015), home automation (Dixon et al., 2012), applying medication in health care settings (Baker, 2012; Herold, 2011), or managing the United States critical infrastructure (Auerswald et al., 2008).

Unfortunately, due to their large code size, use of `C` a memory and type unsafe language, complex nature of operation, and poor fault isolation, commodity monolithic operating systems are susceptible to attack (Tanenbaum et al., 2006). Researchers have estimated that monolithic operating systems have between 16 to 75 bugs for every 1000 lines of code (Basili and Perricone, 1984; Ostrand and Weyuker, 2002). Findings also suggest that device drivers, which typically represent 75% of kernel code, have bug rates 3 to 5 times higher than mature kernel code (Chou et al., 2001; Tanenbaum et al., 2006). And yet more recent work shows that the core operating system is exploitable as well (Kemerlis et al., 2014; sqrkkyu, 2007; Perla and Oldani, 2010; Argyroudis and Glynos, 2011; argp and Karl, 2009). Given a conservative estimate a typical commodity monolithic operating system with approximately 20 million lines of code will have 320 thousand bugs, a subset of which are exploitable by attackers as vulnerabilities.

The types of exploits operating systems suffer from include traditional problems associated with any type unsafe languages: uninitialized or corrupted pointer dereferences, memory safety errors, integer overflows, race conditions, logic bugs, and return-oriented programming all of which lead to the system performing unintended operations (Szekeres et al., 2013; Perla and Oldani, 2010). These vulnerabilities have led to several real world commodity monolithic operating system exploits (Argyroudis, 2010; LMH, 2006; Apple Computer, Inc., 2005; Microsystems, 2003; Starzetz and Purczynski, 2004; Guninski, 2005; Starzetz, 2004; Starzetz and Purczynski, 2004; van Sprundel; BID, 2014; Cook, 2013a,b; Sowa, 2013).

Once a commodity monolithic operating system is breached, such as in Windows, Linux, or Mac OS X, the attacker has full system privileges, thereby eliminating all information protection guarantees the system may have had, including for otherwise absolutely secure applications. The primary goal of the attacker at this stage, is to use supervisor privileges to perform unauthorized operations and persist in the system. Attacks of this nature, called kernel rootkits in general, seek to violate any of the integrity, confidentiality, or availability such as stealing passwords and secret data in the system. Rootkits also commonly target modifying memory resident data to surreptitiously coexist with legitimate system computation for as long as possible (Kong, 2007). The end result is an infiltrated system with perpetrators that are hard to distinguish amongst the large and complex legitimate operations of the kernel.

An additional threat to monolithic operating systems includes the direct installation of compromised code, either as inserted by malicious insider threats or by installing code as requested in phishing attacks (Dhamija et al., 2006; Falliere et al., 2011). The Linux kernel in one release alone had over 1400 different people commit code, which makes it intractable to evaluate the trustworthiness of the resultant operating system: if only one of these developers is rogue then the whole system is compromised. Similarly, phishing attacks can inject and install malicious code into running operating system kernels, which make prevention extremely challenging assuming that end users may not make the most security conscientious decisions.

The fundamental issue is that popular and widely used commodity operating systems employ a monolithic design, which directly violates all the best known principled design considerations for the most important component in our software stacks. One way to address the security problems of commodity operating systems is to employ an alternative more safe design. Microkernel operating system design satisfies the constraints that are desirable of an information protection system: principled design with a simple and small trusted computing base. However, despite the fact that microkernels are in the midst of a renaissance, with great strides in design, implementation, performance, and even formal verification, the truth remains that commodity monolithic operating systems will not be supplanted overnight due to two key factors: 1) the vast number of deployed devices and 2) the longterm investment in these operating systems that has been estimated at over 5 Billion dollars for Linux (Licquia and McPherson, 2016), which leads to highly tested and stable systems. Therefore, an attractive way, for the near term, is to ensure better security and protection of user information by retrofitting separation and protection into existing monolithic operating systems. This dissertation addresses the fundamental design flaw of monolithic operating system by reorganizing various elements of monolithic operating systems to impose small abstraction changes that restrict access to critical information resources.

Overall, it is paramount that monolithic operating systems be enhanced so that we are not reliant on what has been shown and continues to be an exploitable yet assumed trusted base of operations. If we can identify methods for mitigating external attacks and restricting the damage that can be done by persistent kernel level threats then we can provide greater information assurance for billions of users worldwide.

1.1 The Challenge of Separation in Monolithic Operating Systems

Critical information protection design principles, e.g., fail-safe defaults, complete mediation, least privilege, and least common mechanism (Saltzer and Schroeder, 1975; Saltzer, 1974; Organick, 1972), have been well known for several decades. Unfortunately, commodity monolithic operating systems, like Windows, Mac OS X, Linux, and FreeBSD, lack sufficient protection mechanisms with which to adhere to these design principles. As a result, these operating system kernels define and store access control policies in main memory which any code executing within the kernel can modify. The impact of this default, *shared-everything* environment is that the entirety of the kernel, including potentially buggy device drivers (Chou et al., 2001), forms a single large trusted computing base (TCB) for all applications on the system. An exploit of any part of the kernel allows complete access to all memory and resources on the system. Consequently, commodity operating systems have been susceptible to a range of kernel malware (Kong, 2007) and memory corruption attacks (argp and Karl, 2009). Even systems employing features such as non-executable pages, supervisor-mode access prevention, and supervisor-mode execution protection are susceptible to both user level attacks (Kemerlis et al., 2014) and kernel level threats that directly disable these protections.

There are two primary methods for defending against such attacks: 1) hardening of the kernel to memory corruption and 2) containing attacks that succeed. For external entry point attacks the challenge is that critical control-flow data, *i.e.,* data used in controlling the execution of the system, is directly modifiable via common and powerful memory corruption exploits. This means that to thwart attacks, protection must be inserted to stop memory corruption or at least minimally protect control-flow data. With respect to attack containment and insider threats there is absolutely zero separation and therefore no protection at all: so once the compromise occurs all security is compromised.

One way to address the problem of attack containment is to decompose the single monolithic system; in other words, compartmentalize the system to reduce the amount of components that must be trusted for any particular operation to be secure. Compartmentalization has the effect of minimizing the trusted computing base (TCB) while applying the principles of privilege separation and least privilege (Saltzer and Schroeder, 1975; Lampson, 1974; Lampson et al., 1991; Lampson, 1971; Graham, 1968; Dijkstra, 1968). By partitioning the system and enforcing isolation between subsystems, errors can only impact operations of the containing subsystem, and therefore minimize the potential for both full system compromise and reliability failures. The technique of compartmentalization and privilege separation has not only been suggested as fundamental to the protection of information in computers but has also been used to protect information outside of computers as exemplified by military “need-to-know” access controls (formalized for use in computing systems Bell and LaPadula (1973) and Denning (1976)) and by the Byzantine empire to hide information on its state of the art weapons, “Byzantines compartmentalized knowledge of their system so that no one likely to fall into enemy hands would carry more than a fraction of the secret” (Roland, 1992).

Protection refers to the mechanisms and methods that control access between threads of execution, called principles, to stored information including CPU, memory, and device state; a definition informed by seminal operating systems research from the 1960s and 1970s (Graham, 1968; Dijkstra, 1968; Needham, 1972; Saltzer and Schroeder, 1975; Lampson, 1974; Popek and Kline, 1975; Rushby, 1981). Protection can be enforced by any number of mechanisms such as virtualization hardware as used in paging or segmentation, software fault isolation (Wahbe et al., 1993), or hardware privilege levels (Graham, 1968; Schroeder and Saltzer, 1972). Regardless of mechanism, the goal of information protection systems is to isolate the computation and data of mutually distrusting principles so that the security triumvirate of confidentiality, integrity, and availability are upheld. Traditionally, protection is configured and enforced by the system *supervisor*, *i.e.,* the operating system; however, this approach does not work when the system being protected is the same system that manages the protection system.

Fundamental to enforcing any type of protection in computing systems is the complete separation of the protection mechanisms from the system being protected, or as defined by Saltzer and Schroeder the fundamental design principle of *complete mediation*, “Every access to every object must be checked for authority.” (Saltzer and Schroeder, 1975). If the protection mechanism is not fully separated from the general system then it cannot guarantee separation of malicious kernel principles. *Therefore, to defend against memory corruption as well as compartmentalize commodity monolithic operating systems an isolated protection mechanism must be employed that is separated from and able to enforce complete mediation on the rest of the system.*

Unfortunately, all known protection mechanisms are unsuitable for enforcing intra-kernel protection domains in commodity monolithic operating systems. Microkernels (Tanenbaum et al., 2006; Accetta et al., 1986; Liedtke, 1995; Bershad et al., 1995; Klein et al., 2009; Shapiro et al., 1999) and virtual machine monitors (VMMs) (Xiong and Liu, 2013; Xu et al., 2007; Criswell et al., 2009; Payne et al., 2008; Sharif et al., 2009) employ the combination of separate address spaces and multiple hardware privilege levels to enforce privilege separation. Although microkernels present a more secure design than monolithic operating systems they eschew monolithic operating system design and therefore will not reduce the threat. VMMs suffer from both performance issues and a lack of semantic knowledge at the OS level to transparently support protection easily; also, generating semantic knowledge has been shown to be circumventable (Bahram et al., 2010).

Another technique to split existing commodity operating systems into multiple protection domains uses memory virtualization only, *e.g.,* page protections (Swift et al., 2005). Even though these systems partition monolithic operating systems they only provide reliability because they do not isolate the protection mechanism itself, and therefore are susceptible to an attacker that bypasses the protection system by adding *unmediated* virtual address translations.

Tagged memory architectures support fine-grained word level memory protections (Witchel, 2004; Witchel et al., 2005; Zeldovich et al., 2008). These system modify processor design to add memory tagging architecture framework that operates similar to traditional paging based virtualization. Despite reducing the TCB, these approaches modify the hardware making them not applicable for securing today's commodity monolithic operating systems.

Alternatively, several approaches use software fault isolation (SFI) to create intra-kernel protection domains for securing device drivers (Erlingsson et al., 2006; Mao et al., 2011; Castro et al., 2009); however, they incur high overhead and continue to trust "core" kernel code, which may not be all that trustworthy (argp and Karl, 2009; Criswell et al., 2007, 2009).

In addition to an isolated protection mechanism, privilege separation also requires other essential and challenging features for enforcing isolation between principles in a single-address space system:

- trusted usage of protected submodules,
- authorization for principle and domain identification,
- control of entry points (Saltzer and Schroeder, 1975; Lampson, 1974; Witchel et al., 2005),
- and privilege separation that can be feasibly retrofitted to monolithic kernels.

Protected submodules are shared code that can be called as services, and therefore must be able to operate on data given by one protection domain and provide safety and performance. Authorization is required so that the system can enforce meaningful controls on memory and system access as well as controlled entry points. Entry points are critical so that an attacker cannot violate another modules computation by directly bypassing some amount of its code. Lastly, we must be able to retrofit legacy operating systems so that the approach can be directly applied where it is most needed, commodity monolithic operating systems. In this work we primarily focus on the development of an isolated mediation mechanism, controlled entry and exit into a small subset of kernel code, and retrofitting into a commodity monolithic operating system.

Research Question: In summary, the fundamental research question that this dissertation addresses is how to retrofit efficient protection mechanisms and explicit intra-kernel separation abstractions into commodity monolithic operating systems design and implementation with minimal code changes and runtime overheads? This dissertation addresses this question through a general purpose new operating system architecture called the Nested Kernel and a specialized microarchitectural return integrity mechanism called SecRet to efficiently mitigate challenging operating system attacks.

1.2 General Purpose Protection: The Nested Kernel Architecture

To address the lack of protection mechanism that can be employed for intra-kernel memory protections, we present a new operating system organization, the **Nested Kernel Architecture**, which restricts MMU control to a small subset of kernel code, effectively "nesting" a memory protection domain within the larger kernel. The key design feature in the Nested Kernel Architecture is that a very small portion of the kernel code and data operate within an isolated environment called the *nested kernel*; the rest of the kernel, called the *outer kernel*, is untrusted. The Nested Kernel Architecture can be incorporated into an existing monolithic commodity kernel through a minimal *reorganization* of the kernel design, as we demonstrate using FreeBSD 9.0. The nested kernel isolates and mediates modifications to itself and other protected memory by 1) configuring the MMU such that all mappings to protected pages (minimally the page-table pages (PTPs)) are read-only, and 2) ensuring that those policies are enforced at runtime while the untrusted code is operating. Although similar to a microkernel, the nested kernel only requires MMU isolation and maintains a single monolithic address space abstraction between trusted and untrusted components.

We present a concrete prototype of the Nested Kernel Architecture, called PerspicuOS, that implements the Nested Kernel design on the x86-64 (Intel, 2014) architecture. PerspicuOS introduces a novel isolation technique where both the outer kernel and nested kernel operate at the same hardware privilege level—contrary to isolation in a microkernel where untrusted code operates in user-mode. PerspicuOS enforces read-only permissions on outer kernel code by employing existing, simple hardware mechanisms, namely the MMU, IOMMU, and the Write-Protect Enable (WP) bit in *CR0*, which enforces read-only policies even on supervisor-mode writes. By using the *WP-bit*, PerspicuOS efficiently toggles write-protections on transitions between the outer kernel and nested kernel without swapping address spaces or crossing traditional hardware privilege boundaries.

PerspicuOS ensures that the outer kernel never disables write-protections (e.g., via the *WP-bit*) by 1) *de-privileging* the outer kernel code and 2) maintaining that *de-privileged* code state by enforcing lifetime kernel code integrity—a key security property explored by several previous works, most notably SecVisor (Seshadri et al., 2007) and NICKLE (Riley et al., 2008). PerspicuOS *de-privileges* outer kernel code by replacing instances of writes to *CR0* with invocations of nested kernel services and enforces lifetime kernel code integrity by restricting outer kernel code execution to validated, write-protected code. In this way PerspicuOS *creates two virtual privileges within the same hardware privilege level*, thus virtualizing ring 0.

By isolating the MMU, the Nested Kernel Architecture can enforce *intra-kernel* memory isolation policies that trust only the nested kernel. Therefore, the Nested Kernel Architecture exposes two *intra-kernel* write-protection services to kernel developers: write-mediation and write-logging. Write-mediation enables kernel developers to deploy security policies that isolate and control access to critical kernel data, including kernel code. In some cases, data may require valid updates from a large portion of the kernel, making it hard to protect kernel objects in place, or otherwise not have an applicable write-mediation policy; consequently, we present the write-logging interface that ensures all modifications to protected kernel objects are recorded (a design principle suggested by Saltzer and Schroeder (1975)).

To demonstrate the benefit of the write-mediation and write-logging facilities—for enhancing commodity OS security—we present three intra-kernel write-protection policies and applications. First, we introduce the write-once mediation policy that only allows a single update to protected data structures, and apply it to protect the system call vector table, defending against kernel call hooking (Kong, 2007). In general, the write-once policy presents a novel defense against non-control data attacks (Chen et al., 2005). Second, we introduce the append-only mediation policy that only allows append operations to list type data structures, and apply it to protect data generated by a system call logging facility. Additionally, the system call logging facility guarantees invocation of monitored events, a feature made possible by PerspicuOS's code integrity property, and therefore supports a pivotal feature required by a large class of security monitors (Payne et al., 2008; Sharif et al., 2009). Third, we deploy a write-logging policy to track modifications to FreeBSD's process list data structures, allowing our system to detect direct kernel object manipulation (DKOM) attacks used by rootkits to hide malicious processes (Kong, 2007).

We have retrofitted the Nested Kernel Architecture into an existing commodity OS: the FreeBSD 9.0 kernel. Our experimental evaluation shows that this reorganization requires approximately 2000 lines of FreeBSD code modifications while significantly reducing the TCB of memory isolation code to less than 5000 lines of nested kernel code. Our prototype also demonstrates that it is feasible to completely remove MMU modifying instructions from the untrusted portion of the kernel while allowing it to operate in ring 0. Furthermore, our experiments show that the Nested Kernel Architecture incurs very low overheads for relatively OS-intensive system benchmarks: $< 1\%$ for Apache and 2.7% for a full kernel compile.

1.3 Special Purpose Return Address Integrity: SecRet

The Nested Kernel Architecture presents a general purpose mechanism that can be employed to realize both decomposition and operating system hardening techniques. However, one particular security hardening policy, dynamic full system return integrity, is extremely costly to fully secure. The primary challenge is that return based code-reuse attacks are pervasive and persistent while being able to express Turing complete computation for a diverse set of architectures (Roemer et al., 2012). The only technique that effectively mitigates this threat is to enforce a context-sensitive return control-flow policy (Carlini et al., 2015) that is typically obtained by providing memory safety for all return addresses in the system, which is prohibitively costly in practice.

Therefore, this dissertation also investigates whether or not the abstraction of *independent returns* can be fully removed from all software visibility and what challenges remain. Specifically, we investigate a new microarchitectural defense to operating system memory corruption attacks that violate return-control-flow data in order to corrupt and hijack system execution. The system, SecRet, combines hardware isolated secure return address stacks with new microarchitectural detectors to transparently force all operating system and application *return paths* to "return-to-sender": the result is that returns are treated as direct control-flow transfers and are completely immutable from all software, including operating system and user code. We implement an FPGA prototype of SecRet that successfully boots Linux, and explore SecRet's compatibility with diverse architectures by partially emulating SecRet for x86-64 and ARM-M0+. Additionally, SecRet extends the ISA to securely integrate common exceptional control-flows, *i.e.,* software exceptions, with the dynamic "return-to-sender" semantics—consequently, SecRet enforces dynamic control-flow integrity for software exception handling. Our evaluation demonstrates that SecRet is not only *practical*—boots Linux and passes 460/461 compiling LLVM test suite programs—but that it is also *effective*—isolates all return data from operating system control—and *efficient*—less than 1% performance and 6% area overheads with an in-hardware stack capacity of 8 entries.

1.4 Contributions

This dissertation includes the presentation of the following artifacts and contributions:

Nested Kernel Architecture an operating system organization that enables the isolation of the memory management protection domain. Contributions include the following:

- A new OS organization strategy, the Nested Kernel Architecture, which nests, within a monolithic kernel, a higher privilege protection domain that enables
- kernel developers to explicitly apply *intra-kernel* security policies through the use of write-mediation and write-logging services.

PerspicuOS: a Nested Kernel Architecture prototype that isolates the MMU (vMMU protection domain) in FreeBSD 9.0 on the x86-64 architecture. Contributions include:

- a novel technique to virtualize a subset of CPU and memory state at a single privilege level, allowing various protection domains to operate at a single hardware privilege level: effectively demonstrating a technique that virtualizes the single supervisor privilege mode.
- a novel method for presenting Lifetime Kernel Code Integrity: mitigating all kernel code injection attacks.
- a 232 size reduction in the TCB of code allowed to directly modify the MMU compared to stock FreeBSD 9.0.

Intra-Kernel Memory Isolation Services and Policies: abstractions for expressing and protecting the integrity of memory within the kernel.

- Write-mediation service employed to assert write-once and append-only policies.
- Write-logging service employed to trace all attempts to eradicate malicious behavior from protected data structures.
- Security Monitor Service: provides guaranteed invocation and isolation of security monitors.
- Allocation strategy for providing fine grained memory protections while using a page-granularity protection mechanism.

SecRet: a return address protection domain that completely isolates return addresses and automatically enforces full system return integrity to make the following contributions:

- The first microarchitectural context-sensitive KRI mechanism, SecRet, that protects user and kernel return addresses from operating system memory corruption attacks while bridging the semantic gap to transparently detect thread creation and switching events.
- The first that we know of full FPGA implementation of any hardware-assisted shadow stack system, and evaluate its compatibility, area overheads, and performance.
- ISA extensions to securely integrate software exception handling with context-sensitive function returns, which also enforces `setjmp/longjmp` and presents the first such solution for `try/catch` CFI.
- We demonstrate the compatibility of SecRet design with two diverse architectures, OR1200 FPGA and an ARM-M0+ simulator.
- We evaluate for the first time compatibility of the context-sensitive return policy and SecRet software exception handling instructions with a large corpus of existing software including Linux boot, scripting languages, and successful execution of 460/461 unique benchmarks from the LLVM test suite infrastructure.

1.5 Organization of Dissertation

This dissertation is organized into four high level parts: Chapters 1–3 present the core problem, motivation, and high level background of separation applied to monolithic operating systems; Chapters 4–8 present the general purpose protection mechanism and abstractions as developed in the Nested Kernel Architecture; Chapters 9–11 present the specialized investigation into removing the indirect return abstraction from all software; and Chapter 12 presents a discussion of the overall findings, future work, and conclusions. In some instances discussions and conclusions are nested within the containing chapter if it is more local in application.

Chapter 2

Attacking Monolithic Operating Systems

Monolithic operating systems are susceptible to a myriad of attacks, including traditional memory safety exploits in addition to persistent operating system specific threats such as kernel malware (e.g., rootkits). Security exploits are diverse. However, with respect to this work there are two predominant classes: kernel **entry point** attacks that breach the barrier between unprivileged and privileged execution, and kernel **malware** that take advantage of supervisor privileges for any nefarious purpose. These specific terms distinguish between the particular system flaw exploited to gain supervisor privileges and the malicious behavior that persists to operate with those privileges. This chapter characterizes these two classes of attacks—their primary goals, methods of exploit, and keys to mitigation—in order to highlight the security requirements of commodity monolithic single-address space OS protection needs.

2.1 Entry Point Attacks: Compromising Execution

The holy grail of attackers is to obtain administrator privileges. Certainly, gaining control of a user level application is desirable, but the generally higher privilege kernel presents access to all information on the system as opposed to a single application. There are three primary types of external operating system compromise: denial of service, information leaks and corruption, and privilege escalation to arbitrary code execution. By *external* we mean attacks that do not originate with supervisor privilege, which means that they could be induced either local users or remotely via network packets with respect to the operating system under consideration. The important feature is that they gain unauthorized access to privileged data while operating at a lower privilege level than the host operating system.

Each of these attacks enable various degrees of compromise, and in this work we assume the maximal attacker ability. In this section we detail the common elements of memory corruption to control-flow hijacking attacks and describe methods used for arbitrary code execution. Table 2.1 presents a common subset of the attacks we consider.

Phase	**Attack**	**Defense**
1	Corrupt Data Pointer	Memory Safety
2	Modify Code Pointer	Code Pointer Intgegrity
3	To Address of Malicous Code	Randomization
4	Use pointer by `RET` or `JUMP/CALL`	Control-Flow Integrity
5	Execute Injected Malicious Code	$W \oplus X$
6	Control-Flow Hijack via Code Reuse	High Level Policies

Table 2.1: Control-Flow Hijack Attack and Defense Model: a subset of attack model presented by Szekeres et al. Szekeres et al. (2013).

2.1.1 Memory Corruption

The common thread to all of kernel level attacks is that they exploit memory vulnerabilities of C to corrupt data in the rest of the system. The fundamental problem with protecting execution integrity, for memory unsafe languages, is the direct modifiability of code pointers through memory corruption vulnerabilities. Memory corruption occurs when the runtime behavior of the system violates a memory error: a few examples include writing beyond the end of an array, *buffer overflow*, modifying an index to an array that can target arbitrary memory, *index exploit*, or *null pointer* bugs that have all been exploited in commodity operating systems (Szekeres et al., 2013; Perla and Oldani, 2010; Argyroudis, 2010; LMH, 2006; Kemerlis et al., 2012). Carlini et al. (2015) explored the minimum environment required to launch memory corruption attacks that lead to control-flow hijacking, and they found that with only a single arbitrary memory corruption bug and common application libraries (*e.g.,* glibc), even advanced mitigation systems can be compromised, such as those employing buffer-overflow detection mechanisms (Cowan et al., 1998; Frantzen and Shuey, 2001) amongst others. Hund et al. (2009) showed how such attacks are capable of exploiting operating systems.

2.1.2 Control-Flow Hijacking

Indirect control transfers (ICTs) (e.g., `JMP`, `CALL`, and `RET`) enable the runtime selection of control flow targets, a paradigm utilized by software to support the abstractions of independent functions and code reuse. Unfortunately, the dynamic nature of ICT target selection and the fact that control-flow *destination addresses reside in the same address space with vulnerable data* makes them a target of attack: by exploiting a memory corruption bug (*e.g.,* a missing bounds check in C/C++ programs) and modifying memory resident target addresses, such as function pointers or return addresses on the stack, attackers can direct control flow to instructions sequences of their choosing. Control-flow hijack attacks operate by exploiting memory corruption vulnerabilities to either modify existing code in place and/or modify code-pointers to target attacker controlled operations. In this section we detail the types of code injection and return-oriented attacks that employ existing kernel code to invoke malicious behavior.

Code Injection: Originally attackers violated memory corruption bugs to inject malicious code into the runtime memory. The attack takes two forms: 1) overwrite existing kernel code so that when executed perform attacker controlled different operations, or 2) inject code anywhere in memory (including typical data structures) and overwrite a code pointer to target the injected code: the traditional buffer overflow attack.

Malicious Code Execution with Code Reuse Attacks Once an attacker can write to any arbitrary location in memory they overwrite code pointers, *i.e.,* values in memory that are directly targeted for control-flow transfers, to launch their attack code. Originally, attackers targeted injected code (van der Veen et al., 2012; Szekeres et al., 2013); however, today's defenses for user level code employ non-executable data and code integrity (*e.g.,* the $W \oplus X$ property) which has largely thwarted traditional code injection attacks. In response, code reuse attacks were developed that, instead of targeting injected code, targeted existing legitimate application code. Examples of code reuse attacks include return-into-libc (van der Veen et al., 2012), return-oriented programming (ROP) (Shacham, 2007), and even kernel level return-into-user attacks (Kemerlis et al., 2014).

Return-Oriented Programming (ROP) In a return-oriented program the stack is modified, either in place or by pointing the stack pointer to an attacker controlled region of memory, so that each stack element points to a small instruction sequence that ends in a `RET`. Once the return is executed, control jumps to the address located at the top of the stack while the stack pointer is moved to point to the next instruction sequence, effectively using the stack pointer as a pseudo instruction pointer and the sequence of instructions ending in `RET` as pseudo instructions. These instruction sequences are combined to form logical operations such as an `ADD`, and are called gadgets (Shacham, 2007). By setting up the stack with gadgets of well defined individual and combined behavior, the system invokes each gadget in sequence, thus enabling the execution of arbitrary computation with the privileges of the exploited application.

Prior work demonstrated that the set of gadgets in common applications (Roemer et al., 2012) and operating systems (Hund et al., 2009) realizes Turing-complete behavior, thereby allowing an attacker to execute arbitrary programs from within legitimate application code[1]. ROP attacks have been shown to be general purpose, working on several architectures: ARM, x86, and SPARC (Roemer et al., 2012), thus motivating architecture agnostic mitigations.

Characterizing Kernel Specific Threats Despite the prevalence of attention to user level exploits for arbitrary code execution, the most influential attacks exploit operating systems, where

[1] We refer the reader to related work for a more detailed description and review of existing return-oriented programming techniques (Roemer et al., 2012).

the attacker can invoke arbitrary computation with supervisor privileges (Kemerlis et al., 2014, 2012; Criswell et al., 2014a; Hund et al., 2009). These attacks not only have the power to violate execution integrity of the operating system, but also have a much more devastating impact for all applications because any single kernel exploit undermines the security of the entire system. Moreover, operating system threats are particularly egregious because programs—and by extension its users—implicitly trust the operating system to prevent malicious actions in the system.

The problem is exacerbated because a single memory corruption bug in the operating system can be used by one application to overwrite any memory in the system, including the stack and memory state of another thread—assuming that these addresses are mapped into the kernel portion of the address space. If the operating system is employing something like supervisor mode access prevention (SMAP) an attacker can still violate return integrity by modifying the MMU so that the attacker has access to launch a ret2dir attack (Kemerlis et al., 2014). Thereby transiently circumventing the protection operating systems are deployed to support.

2.2 Persistence and Insider Threats

Once arbitrary supervisor execution is obtained, attackers typically install persistent kernel malware, which focuses on surreptitiously violating data integrity and confidentiality while hiding its presence. Attacks of this nature are typically called rootkits. However, we use the term kernel malware instead of *rootkit* because malware is a more general term referring to the larger class of all potential kernel attacks including those from insider threats or phishing attacks. **Rootkits** are a particular type of kernel malware focused on stealth, distinguishing itself from other types of potential malware behavior (*e.g.,* spreading as a virus).

Insider threats, a form of kernel malware, have the same goal as rootkits: to persist a kernel level threat to violate integrity and confidentiality of data. As such the only distinguishing feature between an insider persistent threat and a rootkit is the entry method. For an insider threat to gain control the insider must get the attack code committed to the operating system under attack. Typical attacks of this nature include Trojans and Jekyll attacks that masquerade as normal functionality, but under certain conditions will execute the malicious behavior.

Finally, phishing attacks seek to lure unsuspecting users into installing the code directly into the operating system. Stuxnet, a famous attack that got past several level of extremely harsh conditions for any remote attack, employed a phishing attack combined with a rootkit and worm to propagate to the intended attacker target. Again the only key distinction between phishing and rootkits is that phishing attacks are directly installed by users.

2.3 Discussion

The overall goal of this dissertation is to explore memory protection mechanisms and separation to mitigate both external memory corruption and persistent threats. In particular, we present a defense against kernel level ROP attacks that requires protected memory regions for return address pointers (Chapter 9) and several classes of rootkit attack defenses (Section 6.3). These mitigations are informative in identifying the potential of the Nested Kernel Architecture for providing directly valuable intra-kernel security protections.

Chapter 3

Monolithic Operating System Security and Resiliency

The goal of an information protection system is to protect information from unauthorized modification, access, and denial of access: the security triumvirate of confidentiality, integrity, and availability. In an operating system this means protecting various elements of user and application state (*e.g.,* documents, cryptographic keys, encrypted data) while in runtime memory and processor state (*e.g.,* data in CPU registers) as well as in long term persistent storage (*e.g.,* disk). Unfortunately, for monolithic operating system design security critical information is directly accessible to any kernel level threat. This fact means that what have become common and trivial memory safety violations violate all security. Therefore, the primary challenge for improving security of commodity monolithic operating systems is to establish efficient and isolated information protection mechanisms that can be deployed on commodity monolithic operating systems.

The goal of this dissertation is to consider the runtime protection of information in the system, and therefore focuses solely on protecting dynamic CPU and memory state. Therefore, in this chapter we review literature with respect to retrofitting protection on and into monolithic operating systems including alternative operating system designs as replacements for monolithic operating systems. Because the core focus of this dissertation is to retrofit protection and separation into monolithic operating systems, we then present a review of existing state-of-the-art protection mechanisms, and conclude with a discussion of their use to provide mitigations (hardening) to external attacks and for providing resiliency through decomposition.

3.1 Monolithic Operating System Design Analysis

The primary flaw of existing commodity operating systems is that they employ a single-address space environment for all supervisor privileged code. Despite the value for performance, simplicity, portability, and extensibility, this design not only inherently presents a large attack surface but also allows diverse, fault-prone code full authority in the system. The most valuable form of assurance for the security of a system would be to formally verify that the code only allows authorized access. Unfortunately, such an approach is intractable for code bases with the complexity and size of commodity monolithic operating systems.

Instead, seminal operating system research proposed a set of fundamental design principles for information protection systems:

Least Privilege: Quite possibly the most impacting and fundamental principle with regard to threat containment is the principle of least privilege (oftentimes referred to in literature as the principle of least authority POLA, we use least privilege in this text), which is to restrict each component in the system to only the information necessary to complete its job (*e.g.,* need-to-know). In this way the potential of successful attacks is restricted to the containing compartment.

Economy of Mechanism: Operating System design should present the simplest possible form so that it can be reasoned about either informally or formally, but it is clear that to vet all potential threat vectors simplicity is the goal.

Fail-safe Defaults: When designing a protection system the default configuration should restrict access instead of permit it. In this way the system starts in a secure state and only explicit access requests modify the availability of authority to access information.

Complete Mediation: To assert protection, all information access must be mediated or checked for valid authorization, otherwise there can be no security at all. Furthermore, it is typical for systems to cache mediation checks (*e.g.,* by configuring the MMU as one example), therefore it is also necessary that under such caching the system must take extra consideration to ensure context dependent mediation.

Separation of Privilege: If possible it is best to separate privilege amongst multiple principles so that the failure of one protection subsystem does not compromise the entire system.

Least Common Mechanism: Minimize the mechanism that must be shared by all elements in the system. The more sharing of mechanisms the greater a central point of failure becomes and degrades security if exploited.

Ease of Use: Security must be easy to use, otherwise it will not be adopted, benefiting no one.

Secure Auditing: The primary consideration in secure system design is to eliminate threats by design, however, some may get by the outer defenses, and therefore secure auditing is critical to recreating and detecting compromise.

In considering these design principles, it is clear that commodity monolithic operating systems not only violate these principles, but in many instances provide the worst case scenario for them. For example, monolithic operating system design is the exact opposite of good fail-safe defaults as any kernel component has full system access, which also leads to zero least privilege separation. Furthermore, commodity operating systems are large and complex, thereby violating principles of simplicity in mechanism, which is further exacerbated by the fact that the entirety of the operating system is shared by every application on the system, which absolutely violates least common mechanism.

In addition to these design principles, several researchers considered the notion of secure supervisor design (Neumann, 1986; Popek and Kline, 1978; Rushby, 1981; Lampson, 1974, 1971, 1983; Dijkstra, 1968; Parnas and Siewiorek, 1975). Among these explorations are derived the basic principles of encapsulation via layered design, as well as the need for hierarchical, kernelized designs to address information flow properties that are desirable of the information protection system to achieve confinement (Lampson, 1973; Bell and LaPadula, 1973). Additionally, these considerations point out that for a secure system, minimality and the most primitive operations of the hardware should be encapsulated in a standard operating system virtualization approach (Lampson, 1971).

Overall, these principles represent a small portion of literature on principled software design: suggesting strategies such as composition, modularity, and encapsulation. These are the source of the problem with monolithic operating systems and must be addressed to improve one of the most pervasive elements in our software stacks.

3.2 Operating System Organizations

The Nested Kernel is a new operating system architecture that seeks to address the inherent design flaws of existing monolithic operating systems by providing a micro-evolution of monolithic operating system organization. In this regard the Nested Kernel Architecture is representative of both a new minimal security kernel as well as an overall operating system design focused on decomposition. In this section we detail key related operating system designs and compare them with the Nested Kernel.

3.2.1 Microkernel

Microkernel operating system design employs the combination of memory virtualization (*e.g.,* paging) and CPU privilege levels to modularize and relocate a large portion of operating system functionality out of the trusted computing base (Hansen, 1970; Wulf et al., 1974; Tanenbaum et al., 2006; Accetta et al., 1986; Liedtke, 1995; Bershad et al., 1995; Klein et al., 2009; Shapiro et al., 1999). Although microkernels present a fundamentally more secure design, they are not applicable to improving existing monolithic operating systems security. Additionally, despite the potential for microkernels to gain widespread adoption (seL4 Klein et al. (2009)), without guarantees for their discontinued usage, commodity monolithic operating systems must be protected in present form. Furthermore, microkernels, despite the tremendous TCB reduction relative to monolithic kernels, could benefit from further compartmentalization: a claim supported by 1) the observation of extraneous functionality in the microkernel beyond what is necessary to provide protection services, as well as 2) reflections from the Separation Kernel design which suggests full isolation of the protection mechanism from all orthogonal functionality (Rushby, 1981).

3.2.2 Virtual Machine Monitor

Virtual Machine Monitors (VMMs) were originally created to address the problem of various applications being written for one operating system and users wanting to run multiple operating systems on the same hardware at the same time (Goldberg, 1974; Arpaci-Dusseau and Arpaci-Dusseau, 2015). This approach, although similar to typical user process multiprogramming, is distinct in that the goal of the VMM is to virtualize hardware resources to operating systems, while at the same time emulate hardware expected by the operating system that may not actually be present on the particular system of execution. With regard to the Nested Kernel, VMMs can provide isolation for intra-kernel monolithic operating system separation, however, the primary purpose of a VMM is to virtualize the hardware resources amongst a set of operating systems not to provide isolation and abstractions for intra-kernel usage. This design goal is distinct from the focus of the Nested Kernel Architecture, which is to provide abstractions and primitives to be used internally by monolithic operating systems.

3.2.3 Exokernel and LibOS

An Exokernel operating system design is focused on *removing layers of abstraction* from the operating system so that applications can build custom, hardware specific optimizations (*e.g.,* paging configurations) and eliminate the inflexibility that arises from standard operating system abstractions and interfaces (Engler et al., 1995; Belay et al., 2012). Access to physical resources is encapsulated by library operating systems (LibOSs). Applications can either include standard LibOSs or create custom ones: LibOSs are compiled into the application itself like a typical library. In this way the Exokernel primarily focuses on the secure multiplexing of access to physical resources by untrusted userspace applications.

3.2.4 Nested Kernel

In some sense the Nested Kernel intersects with each of these diverse operating system organizations. The Nested Kernel shares several philosophies with microkernels: a minimal trusted computing base and the core idea of a security kernel focused purely on mechanisms of isolation and not policies. However, the Nested Kernel reduces even further upon the microkernel concept in that it provides a more minimal layer of trust: in contrast to a typical microkernel that presents address spaces, scheduling, and message passing, the Nested Kernel isolates and abstracts only the MMU. In a similar way to VMMs the Nested Kernel abstracts a hardware component to an operating system, and therefore is quite similar. However, the Nested Kernel is only concerned with removing a small set of abstractions and is not concerned with general purpose emulation and simulation of the CPU. With respect to Exokernel design, the Nested Kernel enforces restrictions on supervisor mode code execution; however, the Nested Kernel is doing so on already privileged monolithic operating system source code, and not exporting this functionality to usermode applications. In this since, the Nested Kernel deprivileges code where the Exokernel securely enhances privileges for subsets of code.

Overall, the Nested Kernel is focused on establishing a set of memory isolation abstractions and primitives that can be used for intra-kernel separation, which even though sharing several design elements with these existing operating system organizations, makes it stand apart. This is just the first step in the micro-evolution of monolithic operating system design to a principally more secure and sound organization.

3.3 Protection Mechanisms

The primary goal of the Nested Kernel is to present a mechanism and abstractions with which to support intra-kernel compartmentalization even when assailed by supervisor privileged malicious intruders. Therefore, we investigate related efforts with respect to the ways in which they isolate the protection mechanism including commodity hardware approaches, prototype hardware, and software based solutions. The goal of any protection mechanism is simply to protect the specified state. The Nested Kernel focuses on protecting sufficient state to ensure isolation of runtime execution state, with respect to monolithic operating system this includes CPU as well as memory state. A third state exists, I/O (*e.g.,* long term storage on hard disk); however, this dissertation only considers runtime protections and proposes similar techniques to those used by Hofmann et al. (2013) for encrypting and decrypting I/O.

3.3.1 Commodity Hardware Protection Mechanisms

Memory Virtualization with Multiple Privilege Levels

Traditional Supervisor User Space Separation Typical isolation of protection domains is enforced by the use of at least two CPU privilege levels, supervisor and user, and memory virtualization with segmentation or paging. In this way commodity monolithic operating system isolate process state, and is also the mechanisms used by microkernels: move untrusted code and data into user level processes. Unfortunately, such an approach would not work for the Nested Kernel because all but a very small portion of the kernel ($<$ 5000 LOC) will have to be moved to user level, which would greatly hinder performance and be expensive to modify existing code to operate in user mode processes.

Hardware-Assisted Virtualization With the rise of VMM usage in cloud computing, hardware vendors added support for common and costly software based techniques of isolation. Amongst these include memory, CPU, and I/O virtualization (Uhlig et al., 2005). The core technique is to employ a new microarchitectural level of CPU and memory isolation to allow the higher privileged component, *i.e.,* the VMM, to operate more efficiently than software based approaches that used binary translation and software based shadow page tables. For instance, Intel's VT-x adds extended paging table (EPT) support that adds hardware for nested page tables, an automated hardware page table walk on fault, and virtual process identifiers to avoid flushing the translation lookaside buffer on transitions to and from the VMM. These techniques could be used as a base mechanism to isolate memory in virtual machines, in fact paravirtualization techniques employed by Xen (Barham et al., 2003) use such a technique to ensure that the memory presented to each guest operating system is limited in the host physical memory accessible. The Nested Kernel employs a similar design, however, instead of externally enforcing these properties on the operating system, integrates the design directly into the core of the monolithic operating system.

Intel Software Guard Extensions (SGX) Intel has also considered the issue of completely removing operating systems from application TCBs with the advent of Intel Software Guard Extensions (SGX) (McKeen et al., 2013; Hoekstra et al., 2013; Anati et al., 2013). SGX creates "enclaves" that contain a subset of application code and data, making it impervious to privileged software attacks. The key idea is that the SGX hardware fabric interposes a layer of protection that asserts the data held within the enclave can only be accessed by the code that is also within the enclave. Despite its promise to completely remove the operating system from the TCB, it has been shown that applications rely upon the operating system as a service for many abstractions that undermine the integrity and confidentiality of the applications via IAGO attacks (Checkoway and Shacham, 2013). In response researchers have explored various techniques to protect full legacy applications using SGX, and combining it with LibOSs to avoid IAGO attack susceptibility. Despite these techniques enhancing the application of the *least common mechanisms* principle, the application still trusts the operating system services in the LibOS (Baumann et al., 2014). Alternative approaches, which the author of this dissertation contributed to, Virtual Ghost (Criswell et al., 2014b), provide similar guarantees but without hardware support.

It is unclear how SGX will impact the Nested Kernel, as it was only recently released. Our Nested Kernel implementation enforces a similar functional goal, restricting supervisor memory access to subsets of memory. However, SGX provides a strong isolation boundary in that it is in hardware and defends against certain types of hardware based attacks. From this perspective Nested Kernel can be observed to provide an alternative mechanism for isolation. Another important consideration is that SGX could be used as the mechanism to isolate the various intra-kernel separation domains in Nested Kernel. However, it is unclear at this time how that will work because SGX runs its code in ring3 privilege level (no privileged instructions), which modifies the assumption of the core operating system running at ring0. It also is not clear how well SGX will perform when fine-grained isolation is being used. These are research questions for future work.

Trustzone Approaches TrustZone is an isolation mechanism that effectively adds a single extra bit of control on all memory access representing the secure world or non-secure world where both worlds have a full traditional software stack, *i.e.,* supervisor and user privilege modes amongst other things. The secure world has higher privilege with the ability to read or write all system memory in contrast to the non-secure world that only has access to memory marked non-secure.

Azab et al. (2014) present TZ-RKP, a system that isolates the MMU and provides an external interface to control MMU updates. Both TZ-RKP and the Nested Kernel utilize a similar interface and technique that was applied by Xen's paravirtualization interface (Barham et al., 2003). TZ-RKP has a similar approach to isolating the MMU as in the Nested Kernel prototype; however, TZ-RKP differs in that it uses TrustZone technology for its isolation and only targets an implementation for isolation and guaranteed execution of security monitors embedded into the commodity monolithic operating system. In contrast, the Nested Kernel presents a general architecture and framework with which to not only enforce mechanism as done by TZ-RKP, but also many other types of memory isolation in the kernel. The Nested Kernel also generalizes the exact properties and interface required to virtualize the MMU for *any architecture*, not just using TrustZone, that utilizes an MMU, even if the system lacks hardware privilege isolation. This means that the Nested Kernel can be retrofitted into not only monolithic kernels, but also could be applied to other types of systems such as VMMs (*e.g.,* Xen). I argue that TZ-RKP demonstrates an instance of the Nested Kernel Architecture: where the primary focus is on isolating and mediating updates to the MMU. The ends sought by each are different: Nested Kernel focuses on presenting a general purpose mechanism for decomposition as where TZ-RKP is focused solely on security monitor support.

Memory Virtualization at a Single Privilege Level

Several approaches have attempted to use virtual memory isolation techniques to directly partition monolithic operating systems; Nooks by Swift et al. (2005) provides lightweight protection domains for kernel drivers and modules. Nooks uses the hardware MMU to create protection domains and changes hardware page tables when transferring control between the core kernel and the kernel driver. Chen et al. (1996) introduced the Rio file cache, which isolates the file cache by protecting it using the MMU and write-protection page permissions. To update the file cache the write-protections are disabled, and at all other times the write-protections are enforced thereby protecting the file cache from spurious updates. Even though Nooks and the Rio file cache create protection domains within the kernel for reliability, the protection policies are directly modifiable by an attacker (e.g., by creating an alias to protected resources and bypassing existing policy enforcement) and therefore can be trivially bypassed by an attacker.

3.3.2 Prototype Hardware Protection

Tagged Memory Architectures

Mondriaan Memory Protection (MMP) presents a tagged memory architecture that enables fine-grained byte level memory protections, gaining in efficiency and granularity over page based mechanisms (Witchel, 2004); MMP is used by Mondrix (Witchel et al., 2005), a retrofitted Linux kernel that utilizes MMP to isolate protection domains for a set of device drivers and the file system stack. Despite Mondrix's ability to provide efficient isolation and enforcement of intra-kernel compartmentalization, 1) Mondrix does not use commodity hardware and therefore is inapplicable to today's commodity monolithic operating systems and 2) it is unclear whether or not Mondrix is secure against a malicious adversary. Witchel et al. (2005, §3) presents a memory supervisor that manages updates to page tables, but it is unclear 1) how this is isolated, *i.e.,* with hardware mechanisms?, and 2) whether or not the memory supervisor depends on the active address translations not being bypassed by malicious virtual address translations, *e.g.,* a spurious mapping that obviates protections. The latter point is mentioned by Zeldovich et al. (2008, §6.1) when they compare their approach to Mondrian Memory Protection claiming that Mondrix is unable to mitigate a malicious attacker in the kernel. If true, then Mondrix could gain the necessary separation from the rest of the kernel using the Nested Kernel and is an interesting direction to pursue.

Loki (Zeldovich et al., 2008) employs a similar approach to Mondriaan by using a tagged memory architecture. Unfortunately, Loki uses a microkernel operating system and utilizes a prototype hardware architecture making it inapplicable to today's commodity monolithic operating systems.

Dynamic Hardware Privilege Levels

Configurable Fine-Grain Protection (CFP) (Wentzlaff et al., 2012) presents a new approach to privilege level architectures by allowing various "functional CPU elements" to be dynamically assigned to different per core privilege levels. The privileges they explore include TLB access, DMA engine access, a user network stack, and I/O on-chip network stack. This work identifies an interesting solution to presenting finer granularity hardware protections. However, it is not applicable to commodity monolithic operating systems, and furthermore it is unclear how an operating system would use these features to partition a monolithic kernel.

3.3.3 Software Protection Mechanisms

Several techniques have been developed to address limitations of hardware only isolation mechanisms. These approaches are typically driven by domain or application specific challenges of efficiently isolating execution within typical abstraction boundaries such as a process (Bugnion et al., 2012; Yee et al., 2009; Ford and Cox, 2008; Criswell et al., 2007). There are two key techniques reviewed in this section: language level isolation that enforce isolation through the removal of visibility of certain abstractions (*e.g.,* pointers) and hybrid software-hardware approaches that enforce protection on machine code representations. In this section we review these two protection mechanism. Again, we reiterate that the goal of these protection mechanisms is to restrict a given piece of code from directly (sans mediation) modifying either privileged CPU or memory resident information. Applying these restrictions to supervisor privileged code is challenging given the native authority of any code running at that level of operation.

Language Only Isolation

Type and memory safe languages, through the abstract virtual machine presented to the developers, inherently limit access to only language visible constructs, and therefore by the translation process and the rules of language semantics enforce protection. Operating systems written in type safe languages provide inherent security enhancements for operating system code (Bershad et al., 1995; Aiken et al., 2006; Saulpaugh and Mirho, 1999). While these approaches isolate kernel components and mediate access to critical data structures, they completely abandon commodity OS design. Most importantly these techniques cannot retrofit protections onto C code, instead they propose replacing C with different languages.

Hybrid Language and Machine Code Isolation

When protections are required to restrict either native or C/C++ code, language level techniques fail to protect memory because of pointer operations being first class abstractions in the code. Therefore, dynamic isolation of memory access operations must be explicitly enforced to assert memory sandboxing. Which is the key focus of software fault isolation.

Software Fault Isolation (SFI) originally proposed by Wahbe et al. (1993) inserts checks into binary code that invokes a runtime monitor to restrict memory accesses to the configured policy. For situations where each principle is fully isolated from other components, and therefore the only goal of the protection system is to restrict access to a set of *segments* of the logical address space, SFI performs well. Unfortunately, when applying SFI to a commodity monolithic operating system the sharing patterns are not so isolated, other than for device drivers, as discussed in Section 3.4.2, SFI is not efficient because it must instrument the entire monolithic operating system, which is very large and complex, and without some type of segmentation, instrumenting the whole kernel to provide reasonable privilege separation would be very costly.

One requirement of SFI imposed upon the code being sandboxed is that the SFI security monitor checks cannot be bypassed. Therefore, these techniques also require the use of control-flow integrity, which adds an additional layer of performance costs.

To protect direct access to hardware resources, state-of-the-art approaches modify the code (source or binary) so that sensitive instructions are executed in controlled ways. These properties can be enforced by either executing static or dynamic binary rewriting of the instruction sequences (Bugnion et al., 2012), manually modifying the code so that sensitive operations become calls into trusted security monitors (*i.e.,* paravirtualization Barham et al. (2003)), or compile the native machine code into an intermediate portable instruction set that are controlled by a special translation layer to restrict the resultant instructions (Criswell et al., 2007)). In each of these cases the goal is to either completely remove the lower level machine abstraction or to enforce some partial property on the invocation of those instructions. The security monitor in these instances operates similarly to a LibOS where the LibOS is unmodifiable by the external code.

Criswell et al. (2007) present SVA, a similar approach to languages using a managed runtime, but does so for operating system code programmed in C, and is an example of the third option above. SVA uses a compiler-based virtual machine that inserts a layer of virtualization between the operating system and the SVA hypervisor by refactoring commodity monolithic operating system to call into an SVA library (like a LibOS for hardware interactions) to modify hardware state. This layer of separation allows SVA to assert control over hardware updates via interposing on instructions. SVA also includes SFI sandboxing to obtain memory isolation, and has been combined with memory safety (Criswell et al., 2007) and control-flow integrity (Criswell et al., 2014a) to ensure non-bypassable SFI. Both SVA Criswell et al. (2009) and HyperSafe Wang and Jiang (2010) employ the MMU and the *WP-bit* to prevent privileged system software from making errant changes to page tables. However, these approaches assume the necessity of SFI and CFI, which the Nested Kernel implementation PerspicuOS demonstrates as an unnecessary assumption as well as presents a new overall operating system organization.

Native Client (Yee et al., 2009), VMWare (Bugnion et al., 2012), and Vx32 (Ford and Cox, 2008) use x86-32 segmentation to sandbox application memory access combined with binary translation and rewriting to control sensitive instruction usage that enables the sandbox to maintain isolation. The sandbox restricts the components access to both CPU and memory state, as well as interposes on certain control-flows to ensure isolation of the sandbox as well as other higher level security policies. The key to these techniques is the use of segmentation to isolate the intra-process memory state. Unfortunately, both techniques depend upon the use of segmentation, which has been discontinued in x86-64 and is unavailable on other popular architectures such as ARM.

3.4 Monolithic Operating System Security Policies

This dissertation is concerned with techniques to retrofit protection into existing monolithic operating system design and implementation in order to address both external and internal threats. We propose two directions, hardening to prevent external attacks and decomposition to support resiliency in the face of successful attacks or insider threats. Therefore, in this section we review state-of-the art techniques for mitigating external attacks, which we call operating system hardening, and techniques that address monolithic operating system resiliency through the use of privilege separation (decomposition).

3.4.1 Monolithic Operating System Hardening

Hardening techniques have a rich and varied history: from specific C based exploitation mitigation to kernel malware runtime mitigation and threat detection. The primary commonality between this dissertation and literature is that any guaranteed protection depends upon a tamper-proof security monitor. Therefore in one sense these efforts are distinct approaches to realizing similar security policies as deployed using the Nested Kernel intra-kernel write-protection services (Section 6.3) and SecRet. Alternatively, these reflect policies that are implementable with the Nested Kernel protection mechanism, and therefore also provide an initial aim for the types of policies to explore. In this review we break down hardening techniques into those that assert general high level information protection policy (*e.g.,* trusted measurement of executables) as carried out by security monitors and those that are specialized to preventing external entry attacks.

Security Monitors

Several approaches deploy security monitors using virtualization nested paging hardware support to protect and record certain kernel events: each has drawbacks. Several such approaches place the monitor in the same TCB as the untrusted code, leaving them vulnerable to attack (Microsoft, 2007; Ries, 2005; Tereshkin, 2006). Other systems, namely Lares (Payne et al., 2008) and the In-VM monitor SIM (Sharif et al., 2009), place the monitor in a VMM (using nested paging support) to provide integrity guarantees about the isolation and invocation of the security monitor. These systems suffer from high performance costs (Payne et al., 2008) or assume integrity of the code region (Sharif et al., 2009). VMM-based monitors must also address VMM introspection problems: the monitor does not understand the semantics of kernel data structures (Garfinkel and Rosenblum, 2003; Jain et al., 2014). In the Nested Kernel, security monitors are isolated from the monitored system, can be invoked much more efficiently via direct calls instead of expensive VMM hypercalls, and completely avoid the VMM introspection problem.

Memory Safety

One possible defense to external exploits is to ensure general purpose memory safety properties for operating system code, thereby thwarting the entry-point of malicious activities. Unfortunately, existing memory safety techniques either do not apply to operating system code (Kuznetsov et al., 2014; Necula et al., 2005; Akritidis et al., 2009, 2008), provide only partial memory safety (Zhou et al., 2006; Erlingsson et al., 2006; Mao et al., 2011), require rewriting the operating system in a type safe language thereby nullifying their applicability to commodity operating systems (Hunt et al., 2005; Golm et al., 2002; Bershad et al., 1995), or incur costly performance degradation to enforce full memory safety features on operating system code (Criswell et al., 2007).

Control-Flow Integrity

Both Hypersafe (Wang and Jiang, 2010) and KCoFI employ static CFI to prevent control-flow hijack attacks. HyperSafe applies static coarse-grained CFI on hypervisor code, but lacks adequate protection for various operating system control flow operations. KCoFI, extends CFI to support operating system (kernel) control-flow integrity (KCFI), which combines traditional CFI with kernel level control flow transfer types. Unfortunately, both of these approaches employ static CFI for returns and therefore can be compromised (Shacham, 2007; Roemer et al., 2012; Carlini and Wagner, 2014; Davi et al., 2014b; Göktas et al., 2014; Göktaş et al., 2014)

Kernel Code Integrity

KCoFI (Criswell et al., 2014a), SecVisor (Seshadri et al., 2007), and NICKLE (**?**) provide kernel code integrity. SecVisor and NICKLE also ensure that only authorized code runs in the processor's privileged mode. PerspicuOS enforces the same policies, but also includes a novel memory isolation mechanism and operating system organization, as well as utilizes a more efficient privilege switch mechanism.

3.4.2 Decomposition

Driver Sandboxing

Several previous efforts (Erlingsson et al., 2006; Castro et al., 2009) employ software fault isolation (SFI) and control-flow integrity (CFI) to isolate kernel components. These systems utilize heavy weight compiler instrumentation in addition to address translation policies to isolate kernel components. LXFI (Mao et al., 2011) uses programmer annotations to specify interface policy rules between kernel extensions and the core kernel and inserts run-time checks to enforce these rules. In contrast, the Nested Kernel does not require compiler-based enforcement mechanisms, alleviates the need for kernel control-flow integrity, and removes the core kernel from the TCB.

Nooks (Swift et al., 2005) provides lightweight protection domains for kernel drivers and modules. Nooks uses the MMU to create protection domains and changes hardware page tables when transferring control between the core kernel and the kernel driver. Although Nooks provides reliability guarantees, it does not consider isolation from malicious entities, and therefore is susceptible to attack, as well as does not consider decomposing the core kernel.

Overall, despite demonstrating promising techniques, driver sandboxing neglects to consider decomposing the core operating system kernel, which has been show susceptible to attack (sqrkkyu, 2007; Perla and Oldani, 2010; Argyroudis and Glynos, 2011; argp and Karl, 2009). Furthermore, applying these techniques to the whole operating system will incur much greater runtime performance overheads as exhibited by Criswell et al. (2007).

Virtualizing the MMU

The Nested Kernel Architecture isolates the MMU by modifying the kernel so that all MMU updates are mediated, and exports an interface that is similar to those of related efforts including Xen (Barham et al., 2003), SVA-OS (Criswell et al., 2009, 2014b,a) and `paravirtops` (Wright, 2006). Although the interface is similar, the Nested Kernel Architecture employs different MMU mapping policies to protect and virtualize the MMU, as well as introduces *de-privileging* to isolate the MMU from malicious uses of sensitive instructions. SecVisor employs similar MMU policies to enforce kernel code integrity (Seshadri et al., 2007) as PerspicuOS; however, SecVisor uses special nested paging hardware support that uses implicit traps on certain hardware events, which is both external to the kernel and has higher costs per invocation than PerspicuOS.

Intra-Kernel Isolation Services

SILVER (Xiong and Liu, 2013) and UCON (Xu et al., 2007) specify policy frameworks (similar to mandatory access control) to enforce access control policies on internal kernel objects using VMM hardware. SILVER exports an access control service that is used by the operating system to specify principals and object ownership access policies through the memory allocator, which are then enforced by the VMM. In contrast, the Nested Kernel prototype uses the x86-64 *WP-bit* to provide a memory isolation mechanism on which SILVER access control polices could be overlaid.

Mondrix (Witchel et al., 2005) is a modified Linux version that utilizes Mondriaan tagged based memory protection architecture. Mondrix extends the basic Mondriaan memory protections to provide cross-domain call stacks and single stack permissions, as well as an implementation that partitions Linux into a supervisor memory management protection domain and isolates several Linux kernel modules. The work presented in this dissertation differs in that the protection mechanisms is a complete commodity approach, which requires a much different technique to retrofitting protection into the kernel. Beyond that the fine-grained separation in Mondrix is complementary as similar types of decomposition could be expressed in the Nested Kernel: a topic of future work.

Chapter 4

Nested Kernel Architecture

The challenge of creating multiple protection domains within a kernel (meaning within code that assumes a single address space and hardware privilege level) is to isolate the physical resources used by each domain without disrupting compatibility of the rest of the system.[1] Traditionally, separation is enforced by software running at a higher processor privilege during the translation of virtual to physical resources, as in paging or segmentation (Zeldovich et al., 2008). Therefore, the primary insight and contribution of the Nested Kernel Architecture is to demonstrate how to virtualize a minimal subset of hardware functionality, specifically the MMU, to guarantee mediation and therefore isolation of *intra-kernel* protection domains within the single address space. By virtualizing the MMU, the Nested Kernel Architecture enables a new set of protection policies based upon physical page types and their mappings within the kernel. This section presents the Nested Kernel Architecture overview, our foundational design principles, and describes the versatility of the Nested Kernel as a hardware-architecture portable design.

4.1 System Overview

The nested kernel architecture partitions and reorganizes a monolithic kernel into two privilege domains: the *nested kernel* and the *outer kernel*. The nested kernel is a subset of the kernel's code and data that has full system privilege, and most importantly, the nested kernel has sole privilege to modify the underlying physical MMU (*p*MMU) state. The nested kernel mediates outer kernel modifications to the MMU via a virtual interface, which we refer to as the virtual MMU (*v*MMU). The nested kernel exports an explicit interface, the nested kernel operations, that enables the outer kernel to perform legitimate writes to PTEs, which are typically memory resident data structures, and *p*MMU associated system state, which is typically represented as information in registers (*e.g.,* paging enabled bits in x86-64). The outer kernel is then paravirtualized (Whitaker et al., 2002; Barham et al., 2003), *i.e.,* modified so all privileged operations become function calls

[1] Subsets of the work presented in Part II were originally published by Dautenhahn et al. (2015). Portions of this research were collaborated on and conducted with Theodoros Kasampalis: intra-kernel write-protection allocation services and uses implementation and evaluation; Will Dietz: implementation of trap handling and assembly code and evaluation, John Criswell: design assessment on PerspicuOS and editing; and Vikram Adve: design and editing.

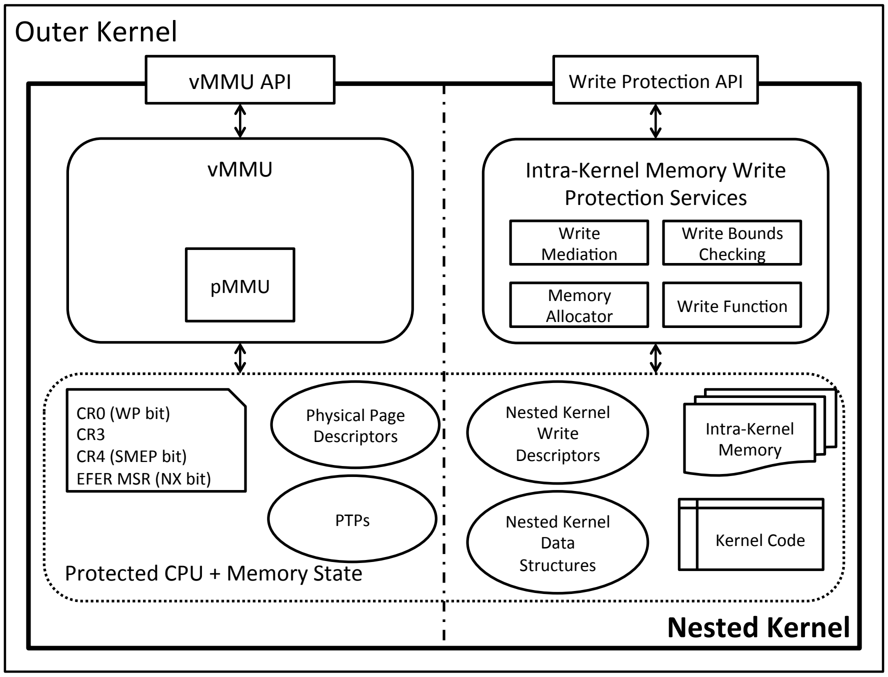


Figure 4.1: Nested Kernel Architecture

into the Nested Kernel, to use the *v*MMU. Figure 4.1 depicts the overall nested kernel architecture: including x86-64 bit CPU state that must be protected.

Similar to previous work (Criswell et al., 2009; Seshadri et al., 2007), the Nested Kernel Architecture isolates *p*MMU updates at the final stage of creating a virtual to physical translation: the point at which a virtual-to-physical translation is made *active* on the processor (i.e., when the processor can use the translation). For example, on the x86-64, address mappings are added to the system by storing a value to a virtual memory location, called a page-table entry (PTE), that resides on a page-table page (PTP) (Intel, 2014). By selecting this abstraction, the outer kernel still manages all aspects of the virtual memory subsystem; however, the nested kernel interposes on all *p*MMU updates, thereby allowing the nested kernel to isolate the *p*MMU and enforce any other access control policy in the system, such as the one used to protect nested kernel code and data. The nested kernel maps the MMU mappings into untrusted kernel space as *read-only*, eliminating the need for the nested kernel to manage the physical page resources while maintaining a shared memory environment for native read speeds.

The Nested Kernel Architecture also protects all kernel code, as well as data selected by kernel developers through the intra-kernel write-protection interface, which is detailed in Chapter 6.

4.2 Design Principles

The Nested Kernel Architecture comprises the mechanism and interface to establish virtual address mappings. As such, we seek to accomplish the following:

Separate resource control (e.g., policy) from protection mechanism (e.g., MMU). We seek the lowest level of abstraction possible to virtualize the MMU, providing only a mechanism that performs updates to virtual-to-physical address mappings. This principle has several benefits: it minimizes the TCB of the privileged domain, maximizes the portability of the nested kernel, and gives maximum flexibility to the types of policies implemented in the outer kernel while maintaining isolation of the nested kernel.

Operating system co-design and explicit interface. OS designers are experts in how their systems work: they represent the best opportunity to enhance the security of the system. Therefore, the nested kernel architecture presents a unified design to realize protections explicitly within the OS rather than transparently enforcing protections via external tools, such as in the case with prior work (Payne et al., 2008; Sharif et al., 2009; Seshadri et al., 2007).

Privilege separation based upon MMU state, not instructions. Traditionally, systems use the notion of rings of protection, where each ring prescribes what instructions may be executed by code in that ring. In contrast, we enforce privilege separation in terms of access to the pMMU, including both memory (e.g,. PTPs) and CPU state (e.g., *WP-bit* in *CR0*).

Minimal architecture dependence. We want to make the Nested Kernel Architecture design as hardware agnostic as possible, assuming only a hardware paging mechanism with page-granularity protections and the ability to enforce write-protections on outer kernel code.

Fine grained resource control. The protections enabled by virtualizing the MMU can be expressed in many ways; we seek to enable fine grained resource control, i.e., protections at byte-level granularity, so that *intra-kernel* isolation policies can be applied to arbitrary OS data structures.

Negligible performance impact. The Nested Kernel Architecture provides isolation and privilege separation without requiring separate address spaces so that it can be applied to operating system architectures with minimal overhead. In our x86-64 prototype, we also run both the outer kernel and nested kernel in the same protection ring (ring 0) rather than via hardware virtualization extensions to avoid costly hypercalls, as evidenced by measurements in Section 7.4.

4.3 Virtualizing the MMU

We summarize the runtime isolation of the *p*MMU as the following property, which Invariants I1 and I2 enforce:

Nested Kernel Property. *The nested kernel interposes on all modifications of the pMMU via the vMMU.*

Invariant 1. Active *virtual-to-physical mappings for protected data are configured read-only while the outer kernel executes.*

Invariant 2. *Write-protection permissions in* active *virtual-to-physical mappings are enforced while the outer kernel executes.*

Active virtual-to-physical mappings are those mappings that may be used by the processor to determine page protections; inactive mappings do not affect memory access privileges. Invariant I2 applies to those processors (such as the x86 (Intel, 2014)) which can disable page protections while still performing virtual-to-physical address translation. While these definitions are independent of whether the MMU uses hardware- or software-managed TLBs, we will assume a hardware-managed TLB to simplify discussion.

On a hardware-TLB system, the Nested Kernel Architecture enforces Invariant I1 by 1) requiring explicit initialization of PTPs, 2) creating an explicit interface to update the page-table entries (PTEs), and 3) configuring all PTEs that map PTPs as read-only. Therefore, any PTP that has not been explicitly initialized at boot time by the nested kernel or declared by the outer kernel via the *v*MMU is rejected from use, enforcing Invariant I1.

Invariant I2 is a particular security policy, which can be enforced by a variety of mechanisms, including single-level page protection mechanisms such as used in our prototype PerspicuOS or alternative mechanisms such as a virtual machine monitor running at a higher hardware privilege level. Section 5.2 details how we ensure Invariant I2 is enforced in PerspicuOS on the x86-64 architecture.

4.4 Portable Mechanisms to Enforce the Nested Kernel

A critical aspect of the Nested Kernel is that it specifies the particular policy of MMU isolation and can be enforced via any *protection mechanism* that adheres to the *Nested Kernel Property*. In this case the Nested Kernel Architecture as a design minimally assumes that read-only policies can be enforced on supervisor code accesses. Much like an Exokernel is typified by providing LibOSs and giving user level applications access to hardware, the Nested Kernel is typified by nesting a single MMU protection domain within larger single address space kernels, and will in the future be typified by the decomposition of monolithic kernels into many intra-kernel protection domains.

The Nested Kernel specifies a high level policy that reorganizes monolithic operating systems so that the outer kernel has a limited view of the MMU, employing encapsulation to restrict access to the real device (Parnas and Siewiorek, 1975). The core focus is to ensure that the outer kernel cannot insert any unmediated mappings and bypass the *Nested Kernel Property*. In architectures such as x86-64 and ARM, mappings are inserted by either adding entries to PTPs or by modifying the base page table pointer (*e.g., CR3* in x86-64) to point at a set of PTPs.

Protecting the PTPs from malicious modification can be enforced by either configuring all mappings to PTPs as read-only (*e.g.,* Hofmann et al. (2013); Criswell et al. (2014b); Barham et al. (2003)) or by utilizing SFI to sandbox all memory manipulating instructions from modifying the protected state (*e.g.,* Wang and Jiang (2010); Criswell (2014)): mechanisms that enforce write protections even for supervisor mode code accesses. Most importantly any mechanism that can enforce the read-only property will correctly implement the Nested Kernel.

Enforcing Invariant I2 amounts to ensuring that the outer kernel cannot gain control of execution while read-only permissions are disabled. One possible mechanism for enforcing I2 is to execute the outer kernel at user level privilege, which has no ability to directly modify any of the privileged MMU state or to directly circumvent control-flow while protections are down: a similar approach taken by microkernels. The Nested Kernel could be implemented using this methodology, but unfortunately if this approach is taken then outer kernel assumptions about running at the higher privilege layer will be violated requiring more code changes and present higher performance costs; this technique would be similar to VMM based isolation using Type-2 hypervisors as used in the VMware hosted architecture (Bugnion et al., 2012).

Alternatively, either hardware virtualization extensions or ARM TrustZone could be used to protect the execution integrity of the nested kernel. In this instance the nested paging support or the secure world address space can be utilized to provide the read-only permissions and the hardware privileged execution zone can be utilized for nested kernel execution. Two independent research systems (Intel, 2015; Wang et al., 2015), released after the original publication of the Nested Kernel, employ VT-x to isolate the MMU in this way. Another employs TrustZone to achieve the same MMU encapsulation (Azab et al., 2014).

Yet another way that I2 could be enforced is by using single privilege level techniques. These techniques are characterized by either introducing language level isolation (*e.g.,* Criswell (2014)) or by similar techniques that we introduce using the *WP-bit* in our prototype Nested Kernel for x86-64. In fact, independent research teams have isolated the MMU on the ARM architecture, and deployed processor level isolation using instruction deprivileging. Song et al. (2016) and Azab et al. (2016) separate PTPs into separate privileged address spaces that are only loaded through special control registers that the higher privilege component controls (TTBR0, TTBR1, and TTBCR). In this way only the privileged component (that resembles the nested kernel) can load the writable mappings for the PTPs and therefore enforce I2, thereby demonstrating Nested Kernel portability to the ARM architecture.

Chapter 5

PerspicuOS: A Nested Kernel Prototype

We present a concrete implementation of the Nested Kernel Architecture, named PerspicuOS, for x86-64 processors. PerspicuOS introduces a novel method for ensuring privilege separation between the outer kernel and the nested kernel while running both at the highest hardware privilege level, effectively creating two virtual privilege levels in *ring 0*. PerspicuOS achieves this goal by taking advantage of x86-64 hardware support for efficiently enabling and disabling MMU write protection enforcement and by controlling which privileged instructions can be used by outer kernel code. More specifically, PerspicuOS applies the design presented in Section 4.3, by configuring all mappings to PTPs as read-only and *de-privileging* the outer kernel so that it cannot disable write-protection enforcement at ring 0. PerspicuOS *de-privileges* the outer kernel by scanning all outer kernel code to ensure that it does not contain instructions that disable the *WP-bit* or the MMU. Additional hardware features (described in Section 5.5) prevent user-space code or kernel data from being used to disable protections.

In this Chapter, we describe our threat model, specify a set of invariants to maintain the *Nested Kernel Property*, and then discuss how PerspicuOS maintains the invariants through a combination of virtual privilege switch management, MMU configuration validation, and lifetime kernel code integrity.

5.1 Threat Model and Assumptions

In this work, we assume that the outer kernel may be under complete control of the attacker who can attempt to arbitrarily modify CPU state. Furthermore, we assume that an attacker can modify outer kernel source code, i.e., that outer kernel code may be malicious. Moreover, we do not assume or require outer kernel control flow integrity, which means that an attacker can arbitrarily target any memory location on the system for execution. For example, since nested kernel and outer kernel code may reside in a unified address space, an attacker could attempt to redirect execution to arbitrary locations within nested kernel code, including instructions that toggle write-protections (i.e., the nested kernel must take explicit steps to prevent such control transfers or render them harmless).

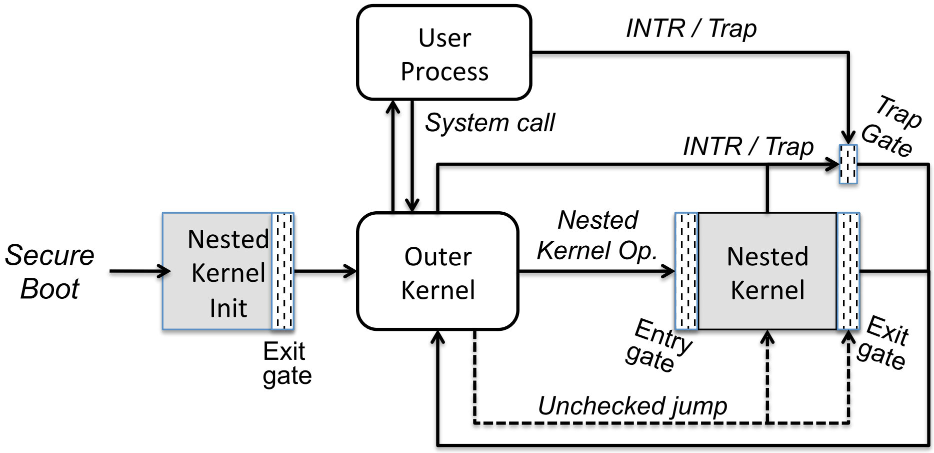


Figure 5.1: PerspicuOS State Transition Diagram. Only shaded blocks can execute PerspicuOS privileged operations (Table 5.1). All transitions out of the nested kernel must go through the Exit Gate.

We assume that the nested kernel source code and binaries are trusted and that the nested kernel is loaded with a secure boot mechanism such as in AEGIS (Suh et al., 2003) or UEFI (Unified EFI, 2010). We also trust mediation functions, a necessary requirement to ensure security checks execute in PerspicuOS. We assume that the nested kernel and mediation functions are free of vulnerabilities, and given the small source code size (less than 5,000 lines-of-code), the nested kernel could be formally or manually verified. Furthermore, we assume that the hardware is free of vulnerabilities and do not protect against hardware attacks.

5.2 Protection Properties and Invariants

The nested kernel design specifies two invariants that must hold to enforce the *Nested Kernel Property*. Invariant I1 requires that all active mappings to PTPs be configured as read-only; Invariant I2 requires that these configurations be enforced while the outer kernel is in operation. We systematically assessed the x86-64 architecture specification (Intel, 2014) to identify both the necessary hardware configurations to realize invariants I1 and I2 and the hardware configurations that may violate those invariants. For example, write-protections are enforced on supervisor-mode accesses when both the *WP-bit* is set and the mapping is configured as read-only; however, alternative execution modes, such as System Management Mode (SMM), can bypass write-protections when invoked. From this assessment, we derive the following invariants that ensure that invariants I1 and I2 hold.

5.2.1 Supporting Invariant I1

The set of active mappings in x86-64 is controlled by the *CR3* register and a set of in-memory PTPs (Intel, 2014). *CR3* specifies the base address of a "top-level" page serving as the root for a hierarchical translation data structure that is traversed by the MMU (Intel, 2014). To ensure that all translations to protected physical pages are marked as read-only (thereby asserting I1), PerspicuOS enforces the following invariants:

Invariant 3. *Ensure that there are no unvalidated mappings prior to outer kernel execution.*

Invariant 4. *Only declared PTPs are used in mappings.*

Invariant 5. *All mappings to PTPs are marked read-only.*

Invariant 6. *CR3 is only loaded with a pre-declared top-level PTP.*

5.2.2 Supporting Invariant I2

PerspicuOS must ensure that, while the outer kernel is operating, MMU write-protections are continually enforced. Read-only permissions are enforced by x86-64 when the processor is operating in long mode with write-protections enabled, i.e., Protected Mode Enable (*PE-bit*), Paging Enabled (*PG-bit*), and Write-Protect Enable (*WP-bit*) bits are set in *CR0*; Physical Address Extensions (*PAE-bit*) bit is set in *CR4*; and Long Mode Enable (*LME-bit*) bit is set in the *EFER* model specific register (MSR) (Intel, 2014). Therefore, PerspicuOS considers scenarios where the outer kernel attempts to 1) disable the *WP-bit* while in operation, 2) disable paging by modifying the *PG-bit*, or 3) subvert control flow of the nested kernel so that the outer kernel gains control of execution while the *WP-bit* has been legitimately disabled for nested kernel operations. PerspicuOS ensures that the *WP-bit* is always set while the outer kernel is in operation and that any instantaneous mode changes that could disable paging, such as an SMM interrupt, are directed to nested kernel control.

Invariants I7 and I8 capture the requirements of the *WP-bit*.

Invariant 7. *The WP and PG flags in CR0 are set prior to any outer kernel execution.*

Invariant 8. *The WP-bit in CR0 is never disabled by outer kernel code.*

When the *PG-bit* is disabled, the processor immediately interprets virtual addresses as physical addresses (Intel, 2014). As Section 5.7 describes, preventing the outer kernel from clearing the *PG-bit* is impossible. Instead, PerspicuOS enforces the following invariant:

Invariant 9. *Disabling the PG-bit directs control flow to the nested kernel.*

Additionally, SMM may be invoked by the outer kernel and therefore, PerspicuOS must also assert control on the SMI interrupt.

Invariant 10. *The nested kernel controls the SMM interrupt handler and operation.*

Given that the previous set of invariants hold, the outer kernel might attempt to manipulate CPU state or outer kernel memory in such a way as to cause control-flow to move from nested kernel code to outer kernel code without re-enabling the *WP-bit*. Therefore, to ensure write-protections are always enforced, PerspicuOS must protect against control-flow attacks on nested kernel execution in two specific cases: interrupt control flow paths and nested kernel stack state manipulation.

PerspicuOS ensures that all exit paths from the nested kernel to the outer kernel enable the *WP-bit* (shown in Figure 5.2b), which is captured in the following invariant:

Invariant 11. *Enable the WP-bit on interrupts and traps prior to calling outer kernel interrupt/-trap handlers.*

Because the trap handlers are a part of the nested kernel, the Interrupt Descriptor Table (IDT) (Intel, 2014) must be placed in protected memory and modifications of the Interrupt Descriptor Table Register (IDTR) must be solely a nested kernel operation.

Invariant 12. *The IDT must be write-protected, and the IDTR is only updated by the nested kernel.*

On a multiprocessor system, code running in outer kernel context on one core could modify the return address stored on the stack by code running in nested kernel on another core if the stack is in outer kernel memory. This would cause nested kernel code to return to outer kernel context without enabling the *WP-bit*. Therefore, PerspicuOS must ensure that code running in the nested kernel uses its own stack located in nested kernel memory.

Invariant 13. *The nested kernel stack is write-protected from outer kernel modifications.*

5.3 System Initialization

PerspicuOS must ensure that all mappings to protected pages (*e.g.,* PTPs, code, nested kernel data, etc.) are configured as read-only and that paging is enabled prior to outer kernel execution, as suggested by invariants I3 and I7. Therefore, PerspicuOS, as depicted in Figure 5.1, initializes the paging system so that invariants I3—where validation implies invariants I4, I5, and I6 by registering all protected pages in nested kernel data structures—and I7 are enforced prior to outer kernel execution by using secure boot and "nested kernel init" functionality, thereby initializing all PTEs in the system.

5.4 Virtual MMU Interface

PerspicuOS provides a set of functions, called the nested kernel operations, that allow the outer kernel to configure the *p*MMU. The nested kernel operations interpose on underlying x86-64 instructions, called *protected instructions*, to isolate the *p*MMU. There are two classes of nested kernel

Operation	**x86 Instruction**	**Description**	**Constraints**
nk_declare_PTP	None	Initialize physical page descriptor as usable in page tables	Asserting invariant I4
nk_write_PTE	mov VAL, PTEADDR	Update *p*MMU mapping	Asserting invariants I4 and I5
nk_remove_PTP	mov VAL, PTEADDR	Remove physical page from being used as PTP	Supporting invariants I4, I5, and I6.
nk_load_*CR0*	mov %REG, %*CR0*	Controls enforcement of read-only mappings	*WP-bit* must be set: invariant I8
nk_load_*CR3*	mov %REG, %*CR3*	Controls MMU mapping base PML4 page	Value must be a declared PML4-PTP
nk_load_*CR4*	mov %REG, %*CR4*	Controls user mode execution with *SMEP* flag	*CR4 SMEP* flag must be 1
nk_load_MSR	wrmsr Value, MSR	Control enforcement of no-execute permissions	*EFER* NX-Bit must be set to 1

Table 5.1: Nested Kernel Operations, *Protected Instructions*, Description, and Constraints

operations: those that control the configuration of the hardware PTPs via memory writes and those that control updates to processor control registers.

The nested kernel enforces *p*MMU update policies by assigning types to physical pages based upon the kind of data stored in each physical page. The page types include PTPs, nested kernel code and data, outer kernel code and data, user code and data, and data protected by the intra-kernel write-protection service. This type information, along with the number of active mappings and a list of all virtual address mappings to the page, is kept in a *physical page descriptor*.

The outer kernel uses the `nk_declare_PTP` operation to specify the physical pages to be used as PTPs. The `nk_declare_PTP` operation takes, as arguments, the level within the page table hierarchy at which the physical page will be used and the address of the physical page being declared, then zeros each page to eliminate any stale data, write-protects all existing virtual mappings to the physical page, and registers the physical page as a PTP by updating the page's physical page descriptor.

Once declared, a physical page cannot be modified directly by outer kernel code. Instead, the outer kernel uses the `nk_write_PTE` operation, which inspects and validates all mappings prior to insertion. The nested kernel uses the previously described physical page type information along with a list of existing mappings to each page to ensure that 1) if the PTE does not point to a data page then it targets a declared PTP and 2) all mappings to PTPs are write-protected, thereby ensuring invariants I4 and I5 respectively. The nested kernel also protects nested kernel code, data, and stack pages to avoid code modifications that would eliminate mediation or functionality of the *p*MMU update process. We also ensure that the update does not write to any kernel data protected by the nested kernel; this is done via a simple check that ensures that the physical page being updated was previously declared as a page table page.

The second group of operations configure the paging hardware itself. We expose an interface for updating *CR3* to ensure that it only points to a declared top-level PTP, called PML4-PTP, thereby ensuring invariant I6. The interface for modifying other registers ensures that paging and lifetime kernel code integrity protections are not disabled by outer kernel code. The description of these mechanisms are in Sections 5.5 and 5.7.

5.5 Lifetime Kernel Code Integrity

To prevent *protected instructions* from being executed while in outer kernel context, PerspicuOS first validates all code before making it executable in supervisor-mode, and second, protects the runtime integrity of validated code by enforcing lifetime kernel code integrity, thereby maintaining invariants I6 and I8. PerspicuOS enforces *load time outer kernel code validity* by scanning binary code to ensure that it does not contain any *protected instructions*, including at unaligned instruction boundaries. Then PerspicuOS enforces dynamic lifetime outer kernel code integrity by configuring the processor and *p*MMU so that 1) by default all kernel pages are mapped as non-executable (enforced by the no-execute bit (*NX-bit*) in the *EFER* MSR), 2) validated kernel code pages are mapped with read-only permissions, and 3) user-space code and data are mapped as non-executable in supervisor-mode by employing supervisor-mode execution prevention (*SMEP* in *CR4*) (Intel, 2014), thereby preventing the outer kernel from executing any *protected instructions* contained within user-mode pages. Note that because protecting the nested kernel depends upon kernel code integrity both *EFER* and *CR4* must also be removed from outer kernel's ability to execute and are thus *protected instructions* as depicted in Table 5.1.

5.6 Virtual Privilege Switches

In PerspicuOS, the nested kernel and outer kernel share a single address space. Therefore, nested kernel operations are essentially function calls to nested kernel functions that are wrapped by entry and exit gates that (among other things) disable and enable the *WP-bit*. Virtual privilege switches occur when write-protection is disabled (which only occurs on nested kernel operations). In this section, we detail PerspicuOS entry and exit gates and describe the ways in which PerspicuOS ensures that the outer kernel does not gain control while write protections are disabled (enforcing I11, I13) and how the gates ensure that mediation functions execute (ensuring I4 and I5).

5.6.1 Nested Kernel Entry and Exit Gates

The nested kernel entry and exit gates ensure that there is a clear and protected privilege boundary between the nested kernel and the outer kernel. The routines depicted in Figures 5.2a and 5.2b perform the virtual privilege switch. The entry gate (Figure 5.2a) disables interrupts, turns off

```
entry:
  pushfq                          Save current flags
  cli                             Disable interrupts
  mov %rax, -8(%rsp)              Spill regs for temps
  mov %rcx, -16(%rsp)
  mov %rsp, %rcx                  Save stack ptr in rcx
  mov %cr0, %rax                  Get current CR0 value
  and ~CR0_WP,%rax                Clear WP bit in copy
  mov %rax, %cr0                  Write back to CR0
  cli                             Disable interrupts
  mov PerCPUSecureStack,%rsp      Switch to secure stack
  push %rcx                       Save orig stack ptr
  mov -0x8(%rcx), %rax            Restore spilled regs
  mov -0x10(%rcx), %rcx
```

(a) Nested Kernel Entry Gate.

```
exit:
  mov 0(%rsp), %rsp               Restore orig stack ptr
  push %rax                       Spill scratch reg
  mov %cr0, %rax                  Get current CR0 value
1:
  or CR0_WP, %rax                 Set WP in CR0 copy
  mov %rax, %cr0                  Write back to CR0
  test CR0_WP, %eax               Ensure WP set
  je 1b                           If not, loop back
  pop %rax                        Restore clobbered reg
  popfq                           Restore flags
                                  (incl interrupt status)
```

(b) Nested Kernel Exit Gate.

Figure 5.2: Nested Kernel Virtual Privilege Switches

system-wide write protections, disables interrupts, and then switches to a secure nested kernel stack; the exit gate (Figure 5.2b) executes the reverse sequence. PerspicuOS by default disables interrupts while in operation; however, we include the second interrupt disable instruction to avoid instances where the outer kernel invokes interrupts that may corrupt internal nested kernel state.

5.6.2 Interrupts

PerspicuOS disables interrupts when executing in the nested kernel. Because the nested kernel is limited to a very small set of functionality, disabling interrupts is not expected to impact performance. Disabling interrupts simplifies the design of nested kernel operations because they can execute atomically: they do not need to contend with the possibility of being interrupted. However, long-running mediation functions may need to run with interrupts enabled—we leave supporting this feature as future work.

PerspicuOS must also ensure that the *WP-bit* is set whenever either a trap occurs or if the outer kernel directly invokes the *WP-bit* disable instruction and subsequently manages to execute an interrupt prior to the second interrupt disable instruction. This is necessary because an attacker could feed inputs to a mediated function that causes it to generate a trap; if the handler runs in the outer kernel, it would be running with write-protection disabled. PerspicuOS protects against these attacks by isolating the x86-64 interrupt handler table (Intel, 2014), enforcing invariant I12, and configuring it to send all interrupts and traps through the nested kernel trap gate first—depicted in Figure 5.1; the nested kernel trap gate sets the *WP-bit* before transferring control to an outer kernel trap handler, following a similar loop as the exit gate starting at assembly label "l" in Figure 5.2b, thus enforcing invariant I11.

5.6.3 Nested Kernel Stack

To enforce invariant I13, PerspicuOS includes separate stacks for the nested kernel. Upon entry to the nested kernel, PerspicuOS saves the existing outer kernel stack pointer and switches to a preallocated nested kernel stack, as shown in Figure 5.2a. When exiting the nested kernel, PerspicuOS restores the original outer kernel stack pointer (Figure 5.2b).

5.6.4 Ensuring Write Mediation

By mapping the nested kernel code into the same address space as the outer kernel, PerspicuOS gains in efficiency on privilege switches; however, the outer kernel can directly jump to instructions that modify the protected state. For example, the outer kernel can target the instruction that writes to PTP entries, thus, bypassing the vMMU mediation. However, such a write will fail with a protection trap because the jump would have bypassed the entry gate, which is the only way to turn off the system-wide write protections enforced by the *WP-bit* (Figure 5.2a). In this way, PerspicuOS ensures that either mediation will occur or the system will detect a write violation.

5.7 Privileged Register Integrity

While the protections in Section 5.5 prevent the outer kernel from directly modifying privileged registers (e.g., *CR0*, *CR3*, *IDTR*), it is possible for the outer kernel to jump to instructions within the nested kernel that configure these registers. To protect against this, the nested kernel unmaps pages containing these instructions from the virtual address space when the outer kernel is executing and maps them only when needed. Invariants I6, protecting *CR3*, and I12, protecting *IDTR*, are enforced using this method because direct modification of these registers can allow the outer kernel to instantly gain control.

While this works for most privileged registers, it does not work for *CR0* (to enforce I8) because the entry and exit gates must toggle write protections, and therefore the instruction to disable *CR0* must be mapped into the same address space as the outer kernel. Therefore, the outer kernel could load a value into the *RAX* register and jump to the instruction in the entry and exit gates that move *RAX* into *CR0*. Ideally, the entry and exit gates would use bit-wise OR and AND instructions with immediate operands to set and clear the *WP-bit* in *CR0*. Unfortunately, the x86-64 lacks such instructions; it can only copy a value in a general purpose register into a control register (Intel, 2014). Note that the *protected instruction* "mov %REG, %*CR0*" is only mapped at three code locations, the entry, exit, and trap gates.

Entry gates do not require verification of the value loaded to *CR0* because the purpose of the entry gate is to disable the *WP-bit*; in contrast, exit and trap gates return control-flow to the outer kernel after modifying *CR0*. The exit and trap gates must therefore ensure that the *WP-bit* is enabled. To do so, PerspicuOS inserts a simple check and loop in the exit gate to ensure that the value of *RAX* has the *WP-bit* enabled, thus ensuring invariant I8. Since these are the only instances of writes to *CR0* in the code, PerspicuOS ensures that outer kernel attacks cannot bypass write-protections by using these instructions.

In the x86-64 architecture, paging is enabled when the processor is in either protected mode with paging enabled (both *PG-bit* and *PE-bit* set) or long mode (PG, PE, PAE, and the LME bits set). To handle the situation where the outer kernel disables the *PG-bit* (regardless of whether the CPU is in long or protected mode), PerspicuOS configures the MMU so that the virtual address of the entry gate matches a physical address containing code that traps into the nested kernel. Therefore, enforcing invariant I9 whenever the *PG-bit* is disabled.

If either the *PAE-bit* or *LME-bit* are disabled while the CPU is in long mode a general protection fault occurs. Because the bits are not updated but instead a trap occurs, the write-protections continue to be enabled and do not require any other solution. According to the Intel Architecture Reference Manual, the *PE-bit* cannot be disabled unless the *PG-bit* is also disabled (Intel, 2014), which is handled by the previously described solution.

5.8 Preventing DMA Memory Writes

The nested kernel must also prevent DMA writes to protected memory. We require that the system have an IOMMU (AMD, 2006) that the nested kernel can use to ensure that DMA operations do not modify any pages protected by the nested kernel.

5.9 Limitations of the Implementation

We implemented PerspicuOS in FreeBSD 9.0. This section describes imitations of our implementation. We did not implement the IDT and IDTR protections. We believe that these will not impact performance as modern OS kernels rarely modify the IDT and IDTR. We ported all instances of writes to MSRs to ensure the NX bit is always set in *EFER*; however, we did not fully implement no-execute page permissions in the PTPs. We do not believe these will negatively impact performance as the nested kernel already interposes on all MMU updates and sets other protection bits accordingly. We also implemented an offline scanner for the kernel binary; we have applied this to the entire core kernel but not to dynamically loaded kernel modules (this is a minor matter of engineering).

Our current implementation uses coarse-gained synchronization even though our evaluation is on a uni-processor. It uses a single nested kernel stack with a lock to protect it from concurrent access. We did not implement protections for DMA writes or enforce nested kernel control on SMI events; however, we do not believe they will negatively impact performance because these are rare events under normal operation. Last, we did not fully implement all features to enforce Invariant I6; however, we did implement code that updates a PTE and flushes the TLB to simulate mapping and unmapping the code that modifies *CR3*. We believe this faithfully represents the performance costs of the full solution.

Chapter 6

Intra-Kernel Write Protection Services

6.1 Write Protection Services API

By isolating the *p*MMU from the outer kernel, the nested kernel can fully enforce memory access control policies on any physical page in the system. For example, the nested kernel can write-protect all statically defined constant data or a subset of system call function pointers that never change at runtime. Therefore, the Nested Kernel Architecture provides a simple, robust API for specifying and enforcing such policies on kernel memory. The write-protection services API, listed in Table 6.1, comprises memory allocation and a data write function with an accompanying byte-granularity mediation policy.

Clients use the *intra-kernel* protection services to allocate regions of memory that are protected by and only written from nested kernel code. When an allocation is requested, either statically via `nk_declare` or dynamically via `nk_alloc`, the nested kernel initializes a write descriptor and allocates an associated memory region. The nested kernel also establishes the memory bounds for the region and sets the mediation callback function (as defined below) that implements the write-protection policy. The nested kernel returns to the client both the write descriptor and virtual address of the newly allocated write protected region, and finally, write-protects all existing mappings to the physical pages containing the memory region.

Clients specify write-protection policies in the form of *mediation functions*. Mediation functions enforce the update policies for write-protected kernel objects, and are invoked by the nested kernel prior to any writes. One example of a simple mediation function is a no-write policy for constant data, with function body, `return false;`, which rejects all writes to the memory region. A more complex example is a write-once policy, such as described in Section 6.3.1, where the nested kernel initializes a bitmap for each byte in the allocated memory region, then upon an `nk_write`, validates that the write is only made to memory not previously written. A significant value of the write-protection interface is that even in the absence of a mediation function (e.g., all writes to the object are permitted), the updates must use `nk_write`, thus thwarting overwrites from memory corruption bugs.

Once a write descriptor, `nk_wd`, is created, the outer kernel executes mediated writes via the

Function	Selected Arguments	Purpose
nk_declare	mem_start, size, mediation_func	Marks all pages RO; initializes an NK write descriptor nk_wd; returns the nk_wd and the pointer to the region.
nk_alloc	size, mediation_func, nk_wd_p	Allocates memory region; invokes nk_declare on it; stores write descriptor in nk_wd; returns nk_wd and pointer to the region.
nk_free	nk_wd	Deallocates memory identified by nk_wd. Memory must have been allocated by nk_alloc. Freed pages can be reused only by a future nk_alloc.
nk_write	dest, src, size, nk_wd	Verifies write bounds; invokes mediation_func, if any; then copies size bytes from src to dest.

Table 6.1: Nested Kernel Write Protection API. nk_declare is for static allocation and nk_alloc is for dynamic allocation.

nk_write function. nk_write operates similarly to a simple byte-level memory copy operation. nk_write performs two checks prior to executing the write: 1) it verifies that the write is within the boundary of the region specified by nk_wd, and 2) it invokes the mediation function, if any. By allowing clients to write only a subset of the memory region, the nested kernel allows protection of aggregate data types without requiring any knowledge about its fields. The interface also makes bounds checking fast by including the write descriptor for constant-time lookup of the descriptor information for the given region.

To fully support dynamically allocated memory, the nested kernel provides nk_free, which deallocates memory previously allocated by nk_alloc. Because an OS exploit could prematurely force nk_free to be called on a memory region and then attempt to store to it, any freed memory must be retained in protected memory, and so we design a simple interface that assumes the allocator is part of the nested kernel.

6.2 PerspicuOS Write Protection Services Implementation

PerspicuOS implements the Nested Kernel Architecture write protection API for write protecting data on physical pages that are marked solely for this purpose and implements the write descriptor instances at *byte-level granularity* to enable multiple concurrent uses of the write service on protected pages while isolating amongst each individual object. One of the primary challenges for implementing the nested kernel write protection services is to devise a method for conquering the protection granularity gap (Wang et al., 2009), specifically the issue of protecting data that is co-located on pages with non-protected data. The nested kernel interface can fully support in-place protections but would result in poor performance: each unprotected object would require a trap and emulate cycle. As such, we elected to focus on schemes that segregate protected data on distinct write-protected pages, separate from unprotected data.

6.2.1 Allocating Protected Data Structures

PerspicuOS presents the intra-kernel write-protection interface as described in Section 6 for allocating and updating write protected data structures. PerspicuOS establishes a predefined ELF memory region to protect global statically-defined data structures. Kernel developers declare write-protected data structures with a C macro that uses a special compiler directive to notify the linker to allocate the object into the specified region. The macro then registers the object into the write descriptor table along with the precomputed bounds and generates both the `nk_wd` and pointer to the region. PerspicuOS provides for dynamic allocation via the interface as described in Section 6. The shadow process list example uses this interface, which is described in Section 6.3.1.

One of the primary challenges of implementing the nested kernel write protection services is to devise a method for conquering the protection granularity gap (Wang et al., 2009), specifically the issue of protecting data co-located on pages with non-protected data. The nested kernel interface can fully support in-place protections but would result in poor performance: each unprotected object would require a trap and emulate cycle. Therefore, we modify the linker script to put this protected ELF region onto its own set of separate pages so that only write-protected data is placed in the region. At boot time, pages belonging to this protected ELF section are write-protected via MMU configuration to ensure the *Nested Kernel Property* for each of these data structures.

6.2.2 Mediation Functions

In an ideal nested kernel implementation, mediation functions would not be in the TCB. This would keep the TCB small regardless of the number of policies and would allow policies to be mutally distrusting. However, to simplify implementation, and to ensure that the mediation functions are executed prior to writes, mediation functions are incorporated into PerspicuOS's TCB. In our evaluation of the write protection interface, we present a set of predefined trusted mediation functions (which, like the mediation functions in an ideal design, do not write to nested kernel memory).

6.3 Enforcing Intra-Kernel Security Policies

The Nested Kernel Architecture permits kernel developers to employ fundamental design principles such as least privilege and complete mediation (Saltzer and Schroeder, 1975). In this section, we explore several intra-kernel security policies enabled by the nested kernel. Our examples demonstrate the nested kernel's ability to combat key mechanisms used by well-known classes of kernel malware such as rootkits (Kong, 2007).

We emphasize that our use cases do not completely solve specific high-level security goals (such as preventing rootkits from evading detection). However, they demonstrate specific key elements for complete solutions. Developing complete solutions is part of our ongoing work.

6.3.1 Nested Kernel Write Mediation Policies

The nested kernel provides kernel developers with the ability to prevent or monitor memory writes at run-time. We illustrate three write-protection policies that this interface can enforce; each can be used for multiple security goals.

Write-Once Data

Several kernel data structures are written to only once, when they are initialized. Other structures are initialized to default values and are only changed once during operation (e.g., the system call table). Our interface can protect these data with very low overhead.

As such, PerspicuOS implements a simple, byte-granularity, *write-once* policy within the nested kernel. It is enforced by maintaining a bit-vector with one bit per byte, initialized to zeroes. When `nk_write` is called, it uses a mediation function that checks whether each bit is set for the memory to be modified; if all the bits are clear, it writes the data and marks those corresponding bits as being written.

We apply the write-once policy to protect the FreeBSD system call table by allocating the table within nested kernel-protected pages and selecting the *write-once* policy, guaranteeing that it can never be overwritten by malware after initialization. This application defends against kernel malware that "hook" system call dispatch by overwriting entries in the system call table to invoke exploit code (Kong, 2007), and could be extended to protect other key kernel code pointers.

Append-Only Data

Operating systems also have append-only data structures such as circular buffers and event logs. These data structures reside in ordinary kernel memory and are vulnerable to kernel exploits, making them unreliable for forensics use.

To protect such data structures, PerspicuOS implements an *append-only* write policy within the nested kernel. It is enforced by maintaining a "tail" pointer to a list structure within the nested kernel. Each call to `nk_write` increments the tail pointer to ensure that writes never overwrite existing data within the region. A stricter policy could ensure that no gaps exist between successive writes. A full solution must also be able to securely write the log to disk when full, which our prototype does not yet do.

We used this policy to implement a system call event logger that records system call entry and exit events in a statically allocated, append-only buffer. System call recording has been a popular target in both research systems (Honarmand et al., 2013; Pokam et al., 2013; Montesinos et al., 2009; Honarmand and Torrellas, 2014) and security monitoring applications (King and Chen, 2003;

Sharif et al., 2009; Hofmeyr et al., 1998; Mutz et al., 2006; Warrender et al., 1999; Payne et al., 2008; Garfinkel and Rosenblum, 2003). However, these systems are susceptible to attack (Wagner and Soto, 2002). By protecting the log buffer, we ensure that rootkits cannot hide traces of malicious system call events and strengthen security staffs' ability to conduct forensics investigations after breakins. Further effort is required to write the logs out to another media for long term storage, and to defend against an attacker that spoofs security events.

Write Logging

A rootkit's primary goal is to hide itself and malicious processes and files. Therefore, they often modify kernel data such as network counters, process lists, and system event logs (Kong, 2007). Some of these data are challenging to protect due to being co-located within large kernel data structures; others cannot be protected by simple write-once and append-only policies. However, the ability to reliably *monitor* writes to such data enables detection of all malicious modifications.

Therefore, we implement a general *write-logging* mechanism that records (and can later reconstruct) all writes to selected data structures. All calls to `nk_write` for a memory region declared with this policy record the range of addresses modified and the values written into the memory. Again, this buffer must be periodically written to disk.

As an example use case of the interface, we use *write-logging* to detect rootkits that attempt to hide processes by corrupting FreeBSD's process list data structure: `allproc`. Instead of logging writes to `allproc` directly, we created a shadow *allproc* data structure that exactly mirrors the original list. Each shadow list entry contains a pointer back to the corresponding `allproc` entry, and any updates to the `allproc` list structure (e.g., unlinking a node) are also performed on the shadow list. More importantly, to fully hide the presence of a particular process from the kernel, the rootkit must use `nk_write` to remove the shadow entry from the shadow list (which is logged).

The logging of shadow list writes enables effective forensics. Security monitors can easily reconstruct the list updates and identify the prior existence of hidden processes. Moreover, we modified the `ps` program to query the shadow list instead of the `allproc` list so that the `ps` program can detect the presence of hidden processes.

6.3.2 Kernel Hardening Properties

The Nested Kernel Architecture can also realize several system security properties because it controls all virtual memory mappings in the system. One example is *lifetime kernel code integrity* (as Section 5.5 explains). This single use case effectively thwarts an entire class of kernel malware (namely code injection attacks). In addition to code integrity, PerspicuOS also marks memory pages as non-executable by default and enables superuser mode execution prevention of user-mode code and data. Even if commodity kernels use these hardware features, they cannot prevent malware from disabling them. PerspicuOS, in contrast, enables these protections *and* prevents malicious code from disabling them.

PerspicuOS can also be used for any type of security monitor that inserts explicit calls into source code to ensure that the monitor both executes and is isolated from the untrusted code. For example, other research has used a Nested Kernel like security monitor to enforce: data-flow integrity protecting authorization and control-flow data in the kernel (Song et al., 2016); kernel code integrity (Azab et al., 2016; Wang et al., 2015); and partial memory safety (Azab et al., 2016; Intel, 2015). Other interesting future directions include: enforcing Kernel Code-Pointer Integrity (Kuznetsov et al., 2014), isolating measurement code to verify binary integrity, create a secure kernel record and replay system (Pokam et al., 2013; Honarmand et al., 2013; King and Chen, 2003), or even to create a secure execution environment (Criswell et al., 2014b).

Chapter 7

PerspicuOS Evaluation

The core contribution of PerspicuOS is to demonstrate a practical instance of the Nested Kernel design with respect to both implementation and performance costs and to reduce the amount of code in the TCB for modifying the MMU. Therefore, we evaluate PerspicuOS by investigating the impact on the TCB, FreeBSD porting effort, and performance overheads. We also investigate the results of the *de-privileging* scanner because the PerspicuOS single privilege level virtualization technique depends upon successfully generating an outer kernel without any *protected instructions*.

7.1 Experimental System Setup

We evaluated the overheads of PerspicuOS on a Dell Precision T1650 workstation with an Intel® Core^{TM} i7-3770 processor at 3.4 GHz with 8 MB of cache, 16 GB of RAM, and an integrated PCIE Gigabit Ethernet card. Experiments requiring a network used a dedicated Gigabit ethernet network; the client machine on the network was an Acer Aspire Revo R3700 with an Intel® Atom^{TM} D525 processor at 1.8 GHz with 2 GB of RAM. We evaluate five systems for each of our tests: the *original* (unmodified) FreeBSD system, the base PerspicuOS, and each of our three use cases: append-only, which is used for system call entry and exit recording; write-once, used for the system call table protection; and write-log, used for the shadow process list. The baseline for the syscall use case was the original FreeBSD modified so that it is logging system call entry and exit events.

7.2 Trusted Computing Base and Kernel Porting

The nested kernel requires porting existing functionality in a commodity kernel to use the nested kernel operations. Our port of FreeBSD to the Nested Kernel Architecture modified 52 files totalling $\sim$1900^{+} LOC changed, including comments. The vast majority of deleted lines were to configuration or build system files—ignoring these, only $\sim$100^{-} LOC were eliminated in the port. Code modifications were measured using Git change logs. We measure the number of lines in the nested kernel with the SLOCCount tool (Wheeler, 2015): the implementation consists of $\sim$4000 C SLOC and $\sim$800 assembly SLOC; the scanner was implemented in 248 python SLOC.

Privilege Boundary	**Time (μsecs)**	**Time / NK Call**
NK Call	0.1390	1.00x
Syscall	0.08757	0.62x
VMCALL	0.5130	3.69x

Table 7.1: Privilege Boundary Crossing Costs.

7.3 Code Scanning Results

To evaluate the feasibility of eliminating all instances of *protected instructions* from the outer kernel, we scanned our compiled kernels and subsequently used manual methods to eliminate all unaligned *protected instructions*. We found a total of 40 implicit instructions for writing to *CR0* (2) and *wrmsr* (38). Most of these instances are due to constants embedded in the code used for relative addressing; therefore, we eliminated them by adjusting alignments, rearranging functions, and inserting *nop*s. A few were due to particular sequences of arithmetic expressions; these were addressed by replacing them with equivalent computation. Finally, a small number of constants in the outer kernel code contained implicit instructions. These were addressed by replacing each constant with two others that were dynamically combined to create the equivalent value.

7.4 Privilege Boundary Microbenchmark

To investigate the impact of different privilege crossings against the Nested Kernel Architecture approach, we developed a simple microbenchmark that evaluates the round trip cost into a null function for each privilege boundary: syscall, nested kernel call, and VMM call (hypercall).

For the syscall boundary experiment we used the *syscall* instruction to invoke a special system call added to the kernel that immediately returns. The VMM boundary cost experiment is performed using a guest kernel consisting solely of *VMCALL* instructions in a loop executing within a VMM modified to resume the guest immediately after this instruction traps to the VMM. The nested kernel cost experiment uses an empty function wrapped with the entry and exit gates as described in Section 5.6.1.

The microbenchmark performs each call one million times and reports total elapsed time. Each microbenchmark configuration was executed 5 times with negligible variance, and the computed average time per call is reported.

Our results, shown in Table 7.1, indicate that a nested kernel call is approximately 3.69 times *less* expensive than a hypercall, thus motivating the performance benefits of implementing the Nested Kernel Architecture at a single supervisor privilege level. User-mode to supervisor-mode calls are faster than nested kernel calls, which take approximately 1.59 times as long.

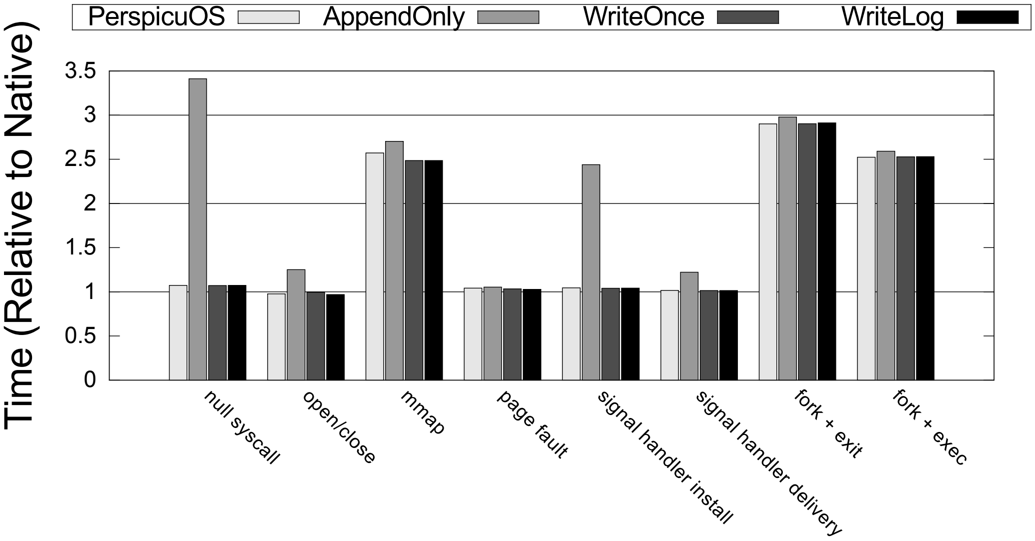


Figure 7.1: LMBench results.

7.5 Operating System Microbenchmarks

To evaluate the effect that PerspicuOS has on system call performance, we ran experiments from the LMBench benchmark suite (McVoy and Staelin, 1996). Figure 7.1 shows the results for our four systems relative to the original FreeBSD. In most cases, our systems are, at most, 1.25 times slower relative to the baseline (unmodified) FreeBSD kernel. `mmap`, `fork+exit`, and `fork+exec`, however, exhibit higher execution time overheads of approximately 2.5 to 3 times. This is because these benchmarks stress the *v*MMU with several consecutive calls to set up new address spaces. Upon investigation, we identified a small set of functions that were responsible for most of this behavior, and preliminary experiments showed a reduction by more than 60% when converting these to batch operations. In the future, we plan to extend the nested kernel interface to allow for batch updates to the *v*MMU in order to reduce overheads for these operations.

We also observe that the write-once and write-log policies incur the same overheads as the base PerspicuOS system, whereas the append-only policy used for system call entry and exit recording incurs higher overheads. In fact, the worst *relative* overhead for this system is the `null syscall` benchmark; it occurs because each null system call makes two nested kernel operations calls.

7.6 Application Benchmarks

To evaluate the overheads on real applications, we measured the performance of the FreeBSD OpenSSH server, the Apache httpd server running on the PerspicuOS kernel, and a kernel compile.

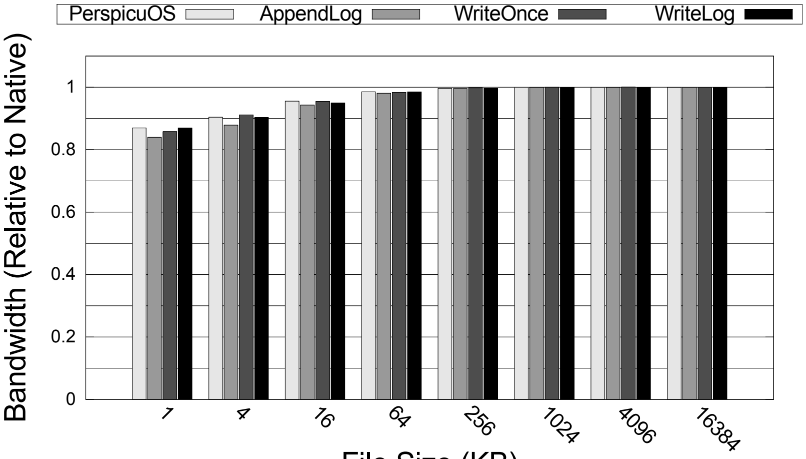


Figure 7.2: SSHD Average Bandwidth.

PerspicuOS	AppendOnly	WriteOnce	WriteLog
2.6%	3.0%	2.6%	2.7%

Table 7.2: Kernel Build Overhead over Native.

We opted to use network servers as they exercise kernel functionality more heavily than many compute bound applications and are therefore more likely to be impacted by kernel overhead.

OpenSSH Server: For the OpenSSH experiments, we transferred files ranging from 1 KB to 16 MB in size from the server to the client. We transferred each file 20 times, measuring the bandwidth achieved each time. Figure 7.2 shows the average bandwidth overhead, relative to native, for each file size transferred. The maximum bandwidth reduction is 20% for 1 KB files. Transferring files above 64 KB in size has less than 2% reduction in bandwidth.

Apache: For the Apache experiments, we used Apache's benchmark tool `ab` to perform 10000 requests using 32 concurrent connections over a 1Gbps network for file sizes ranging from 1 KB to 1 GB. We performed this experiment 20 times for each file size, and present the results in Figure 7.3. The experiment results reveal negligible, if any, overheads that are within the standard deviation error.

Kernel Build: The kernel build experiment cleaned and built a FreeBSD kernel from scratch for a total of 3 runs (the variance was virtually negligible). The worst run times are shown in Table 7.2. The results show an overhead of about 2.6% for the base PerspicuOS, system call table, and process list configurations; and an overhead of 3% for the system call entry/exit case.

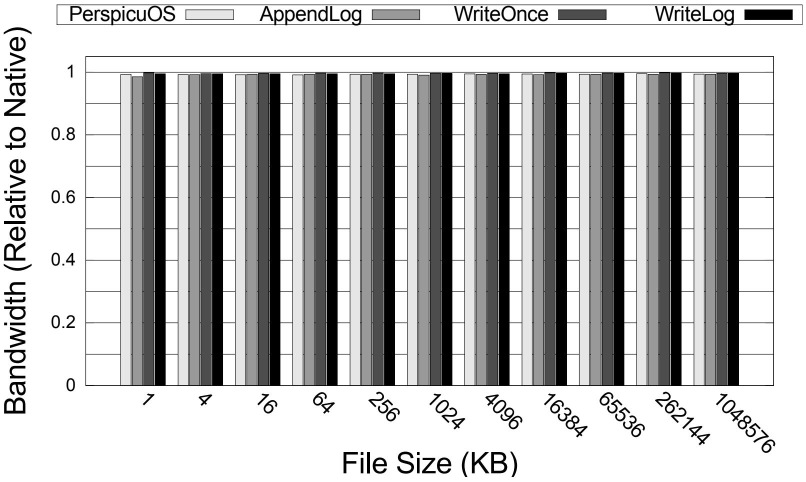


Figure 7.3: Apache average bandwidth.

Chapter 8

Micro-evolution of Monolithic Design

The Nested Kernel Architecture is the first step towards a more secure and resilient operating system design that can be retrofitted directly into existing commodity systems as demonstrated by PerspicuOS. There are several key lessons that we learned in the process of designing and implementing PerspicuOS, the majority of which relate to the challenge of efficiently retrofitting the isolated MMU protection domain with minimal modifications to existing operating system code. We also discovered a new technique that virtualizes x86-64 supervisor privilege, presenting both a new minimal security kernel as well as a foundation with which to deploy efficient security monitors. Overall, we feel this reflects the first step in a *micro-evolution of monolithic operating systems* towards a more principled secure operating system design. In the rest of this chapter we describe the most important lessons we learned and discuss key ongoing and future work to continue the effort of a more resilient commodity operating system design.

8.1 Lessons Learned and Discussion

During the design and implementation of the Nested Kernel and PerspicuOS we learned several important lessons about implementation specific issues as well as the relation between the various artifacts and prior work. In this section we detail key lessons and takeaways.

8.1.1 Virtualizing Supervisor Privilege with the *WP-bit*

PerspicuOS demonstrates a new technique that alleviates the need for higher layers of hardware privilege to gain separation even on supervisor privileged code: restricting access to both CPU and memory state by using simple MMU configuration combined with easy to verify static code properties. This technique impacts both the portability and performance of the resulting solution. Early hierarchical operating system designs (*e.g.,* Multics (Organick, 1972)) had multiple privilege levels; however, due to common hardware platforms only having two privilege levels (*e.g.,* Compaq Alpha and Silicon Graphics MIPS), user and supervisor, operating system designers built their operating systems without any internal separation because the operating system would not be easily portable (Russinovich et al., 2012). In PerspicuOS, virtualizing the single privilege level provides a tremendous benefit in portability in the Nested Kernel design and path to adoption, specially given that the layer of functionality is so minimal: only the MMU.

We believe that the technique introduced by PerspicuOS is unique in that it presents a solution that uses only the MMU combined with simple to verify binary code properties to gain isolation: presenting a mechanism that can be employed in gaining a layer of isolation for future systems. Additionally, using the *WP-bit* eliminates the need to flush the address translation cache on each invocation to the nested kernel, and therefore is more efficient than conventional boundary crossings. Finally, single privilege layer virtualization also has implications for applying similar techniques to VMMs such as Xen where no higher privilege layer exists: a key direction of future work.

8.1.2 Operating System Organizations

The Nested Kernel Architecture is one instance of what early operating system researchers called the "nucleus" of the system: a core component that provides the most primitive functionality and abstractions to service the larger operating system. The microkernel operating system design is one instance of such a *nucleus* and at first glance the Nested Kernel could be consider similarly; however, we argue that the Nested Kernel is an alternative of the same *nucleus* idea, which is distinct from the microkernel. The primary distinction between the Nested Kernel and a typical microkernel is that the nested kernel demonstrates an isolation layer of only the MMU, in contrast to a typical microkernel that provides address spaces, scheduling, and message passing.

Another key distinction that is a result of the Nested Kernel overall approach of retrofitting into monolithic kernels is that microkernels were originally designed as a service and abstraction layer for other operating systems, however, it was developed without any particular operating system in mind. In other words, at their invention microkernels were designed to support a wide variety of potential operating systems, which may be why they are larger than the Nested Kernel. In contrast the Nested Kernel is similarly a primitive nucleus but with a very specific type of operating system in mind: monolithic operating system. Because of this the Nested Kernel can be tailored much more specifically to the abstractions needed to seamlessly, or with minimal effort, integrate into monolithic operating system, while keeping the domains as minimal as possible, such as the case with our design choices of only encapsulating the last step of MMU modification in PerspicuOS. This has the added benefit of leveraging all the stability of a code base with such longevity of implementation and reliability as Mac OS X, Windows, and Linux while moving towards an overall architecture with strong resiliency through decomposition. As described by the ongoing work to extend the basic Nested Kernel services already published, the Nested Kernel and following work will develop and integrate a kernelized design into existing commodity monolithic operating systems.

8.1.3 Performance Sensitivity

Adding layers of security to any code base is going to result in performance degradation, which if unchecked would lead to an impractical option for commodity systems. As our experiments showed, if we had decided to utilize hardware virtualization extensions, the per nested kernel invocation costs would be 3.7 times worse than using the *WP-bit*. Therefore, the use of the *WP-bit* and single privilege level separation not only provides a new virtualization technique but also an efficiently performing choice.

8.1.4 Nesting in the Linear Address Space

One of the biggest challenges of decomposing an existing code base is the cost in terms of code modifications. And thus one of the fundamental design decisions was to maintain as much as possible the linear address space, and *nest* the higher privilege state within the traditional monolithic address-space: in standard operating system parlance, separating addressing from protection. The result is that code outside the nested kernel can still read the protected state without any modifications to memory layout or calls into supervisor code, and only requires entrance into the nested kernel on writes to PTPs, which is a minimal subset of the operating system in the first place. Of course when adding confidentiality these properties may seem less viable, however, when employing confidentiality the kernel designer wants to minimize access, employing least privilege, and therefore external access should be minimal.

8.1.5 The Protection Granularity Gap

A common problem known as the protection granularity gap (Wang et al., 2009) presents itself in the Nested Kernel Architecture write-protection interface. The problem is that memory protections could be defined for only a subset of an entire page of memory; thus, all writes to non-write protected memory on that page will trap. To handle this the Nested Kernel Architecture suggests incorporating a memory allocation scheme that bundles write protected objects into pages set aside for write protection purposes, thereby, avoiding costly trap and emulate to update mixed data pages. Chapter 6 presented such an allocation scheme for both statically and dynamically allocated objects, including examples of its use.

8.1.6 Bridging the Semantic Gap

A common security solution is to use higher privilege components, typically referred to as a security monitor, to enforce secure properties on the monolithic operating system. Examples include code integrity or trust measurements of signed applications (Payne et al., 2008; Sharif et al., 2009; Seshadri et al., 2007), however, existing solutions typically use VMM isolation, which beyond performance has issues of the semantic gap, namely that the data structures in the kernel are not known to the hypervisor. The co-design technique that we employ in the Nested Kernel allows full semantic knowledge of all operating system internal objects and therefore no semantic gap problem. The reason why Nested Kernel conquers this problem is not due to the single privilege level, but rather due to the co-design of the security monitor with the operating system kernel itself.

8.2 Ongoing and Future Work

This dissertation has presented the first step towards a new type of monolithic operating system organization, the Nested Kernel Architecture, that adheres to the fundamental design principles of secure information systems. The overarching theme is to micro-evolve commodity operating system design so that microkernel like properties are incrementally retrofitted *in-situ* in commodity monolithic operating systems. The Nested Kernel has achieved the first step in this direction by retrofitting an isolated protection mechanism and abstraction layer with which to enforce memory and CPU isolation. However, there are several challenging questions that remain, namely how do we decompose existing large monolithic code bases systematically to apply the principles of least privilege? And once we have decomposition policies, how do we efficiently modify the code to adhere to that separation policy while minimally impacting performance? The problem is that of logical separation and runtime protection efficiently and at scale, which is the focus of ongoing research.

Orthogonal and equally important questions arise with respect to specialized security enhancements that defend against particular external attacks. Amongst these include investigating and enhancing PerspicuOS's security by exploring formal verification and type safe implementations, end-to-end security hardening policies such as control-flow integrity or partial memory safety for specific data, and applications of Nested Kernel to cloud environment to investigate 1) security of new containerization technologies, specifically Docker, as well as 2) applying Nested Kernel to decompose popular VMM's. In the rest of this section we detail ongoing work with respect to general purpose, systematic, fine-grained decomposition and protection of intra-kernel components.

8.2.1 Opportunistic Privilege Separation

Separation and protection within monolithic operating systems is necessary to defend against persistent kernel threats (e.g., rootkits) as well as typical memory corruption and control-flow hijacking attacks. In ongoing work we investigate two key questions: 1) how to logically decompose monolithic operating systems (policy) and 2) the abstractions that capture and enforce that decomposition as well as their performance in a real operating system (mechanisms).

First, meaningful decomposition must be general purpose and easily applied, therefore we propose a new decomposition technique called opportunistic privilege separation, which asserts that popular monolithic operating systems have naturally occurring separation despite the inability to fully isolate. We propose that this natural modularity can be opportunistically employed to partition monolithic operating systems into many smaller sized protection domains. In opportunistic privilege separation, kernel objects are logically and automatically partitioned into separate protection domains, and legitimate sharing of data between domains is learned by dynamic tracing to identify valid instances and managed by a runtime monitor.

Second, we introduce microsegmentation, which separates addressing and protection: all kernel objects are uniquely addressed within the single linear address space but access is only permitted based upon the currently executing protection domain—we enforce protections by extending the Nested Kernel and using traditional page-based protection mechanisms. A protection domain is defined by two features: the currently executing thread and the unique kernel code execution point (IP). This allows us to define separation policies based upon the user-level thread the kernel is operating on *behalf-of*, as well as a spatial partitioning of kernel code regions. An example of a spatial domain in Linux is the cryptographic module, which by definition should be well isolated so that only one domain has access to private keys. An example of a temporal domain is a set of processes working closely together but separate from all other processes in the system, which is common in cloud environments such as with Docker.

To minimize decomposition modifications to existing kernel code, the opportunistic privilege separation design modifies existing kernel allocators to permit allocation to specific microsegments, which are mutually exclusive regions of the linear address space: each unique protection domain has its own set of microsegments. When an allocation is flagged as going to a microsegment the allocator introspects the protection domain from the current thread and spatial code segment. This allows for expressive separation policies that capture traditional globally accessible allocations as well as allocations depending on one or both of the thread and code segment.

Protection is enforced by swapping mappings to microsegments on protection domain switches. Controlled sharing is permitted by allowing external protection domains to write to kernel objects and trapping into a security monitor for accesses to mapped out microsegments. On trapping the monitor checks an access control policy for that particular object as learned by the first phase of opportunistic privilege separation. Lastly, our system controls spatial protection domain entry by using in-lined protection-domain switches (similar to the technique used by PerspicuOS). The goal of these techniques is present an incrementally-deployable separation of commodity monolithic kernels.

The opportunistic privilege separation is ongoing research. We have currently developed a kernel memory access pattern tracing tool that tracks modifications to all objects in the Linux kernel. We are collecting and analyzing data in order to learn what naturally occurring boundaries exist withing the Linux kernel.

Chapter 9

Return-to-Sender: Enforcing Full System Return Integrity with Microarchitectural SecRets

One of the most predominant and persistent threats to all C based systems is that of code-reuse attacks, and in particular return-oriented programming. Our work on KCoFI (Criswell et al., 2014a) addressed this issue by employing a technique called control-flow integrity (CFI) that restricts operating system control-flow to only those statically valid flows. Unfortunately, the security policy used in KCoFI was imprecise and thus susceptible to compromise. Enforcing the most precise policy is feasible using either KCoFI or the Nested Kernel for isolation; however, doing so would come at great cost in performance as the frequency of privilege boundary crossings is high for interacting with protected return addresses. Therefore, in addition to providing a new protection mechanism via the Nested Kernel Architecture this dissertation also addresses the challenge of enforcing integrity of full system return control-flows, a policy call kernel-return integrity.[1]

The system presented here, SecRet, defends against return-oriented programming and return-address overwriting by enforcing a context-sensitive function return policy. SecRet does so by embedding an on chip call stack that is automatically updated on function calls and implicitly used for returns. One of the challenges with defending against kernel level return-oriented programming attacks is that the SecRet Stack must be virtualized to allow for stack depth larger than the on-chip buffer, as well as to support multi-threading: the implication being that, while paged off the chip, call stacks are stored in main memory and thus modifiable by an attacker. SecRet isolates this memory using a specialized hardware isolation mechanism that is retrofitted to existing hardware designs. In this chapter we present SecRet, including the specific motivation, problem definition, general system design, and contributions.

9.1 Introduction: Problem and Overview

Operating systems are assumed to be trustworthy, and because they operate with absolute authority within our computing systems, they must live up to that expectation. Unfortunately, commodity

[1] I collaborated on SecRet with Matthew Hicks who is responsible for the FPGA and Arm Cortex-M0+ prototype implementations, evaluations, and writing sub-sections. Matthew also aided in overall editing and review of the chapters in this Part (III).

operating systems are implemented in memory unsafe languages leading to memory corruption vulnerabilities, which attackers use to hijack system execution (Szekeres et al., 2013; Perla and Oldani, 2010; Argyroudis, 2010; LMH, 2006; Kemerlis et al., 2012; Starzetz and Purczynski, 2004; BID, 2014; Cook, 2013a,b; Sowa, 2013). Furthermore, operating systems are more susceptible to attack than a typical application due to their large code base (Tanenbaum et al., 2006), and since they are the apogee of system authority successful attacks eliminate all security guarantees on the system, even by otherwise secure user applications. For our computing systems to be protected we must provide effective mitigations to operating system memory corruption and control-flow hijacking attacks.

One of the most appealing mitigations to control-flow hijack attacks is to prevent execution of malicious code by limiting system behavior to pre-computed control flows (Kiriansky et al., 2002). Static Control-Flow Integrity (CFI), as originally proposed by Abadi *et al.* (Abadi et al., 2009), restricts control-flow transfers to locations as predicted by the program's control-flow graph (Wagner and Dean, 2001; Wagner, 2000; Giffin et al., 2004, 2002; Abadi et al., 2009). Both KCoFI (Criswell et al., 2014a) and HyperSafe (Wang and Jiang, 2010) enforce static CFI for system level code, and when composed with kernel code integrity and data execution prevention ($W \oplus X$), KCFI is generally believed to provide a comprehensive control-flow hijack defense.

Unfortunately, static CFI eschews key characteristics about function *returns* that are only dynamically available. Specifically, static CFI identifies the set of valid return locations of a function as all callers in the statically compiled source code: for example, popular functions, such as `printf`, have a myriad of statically valid callers but only one dynamically valid return location. In contrast, a fundamentally more precise and restrictive CFI policy, *context-sensitive return integrity*, is to restrict returns to the dynamic caller (Giffin et al., 2004; van der Veen et al., 2015): a point supported by Abadi *et al.* (Abadi et al., 2009) as well as recent research by Carlini *et al.* that demonstrates **"evidence that CFI without a shadow stack is broken."** (Carlini et al., 2015) (shadow stacks are a popular approach to enforcing context-sensitive function returns (McGregor et al., 2003; Lee et al., 2003; Ozdoganoglu et al., 2006; Xu et al., 2002; Sinnadurai et al., 2008; Kayaalp et al., 2012)). We conclude that unless existing mitigations make a radical departure from statically derived CFI policies they will remain fundamentally exploitable by control-flow hijacking attacks and undermine all solutions based on CFI. The implication of this finding is that existing operating system CFI approaches, KCoFI (Criswell et al., 2014a) and HyperSafe (Wang and Jiang, 2010), and furthermore all CFI based mitigations, are unsuitable to defend against common return path violation exploits (Cheng et al., 2014; Pappas et al., 2013; Zhang and Sekar, 2013; Fratric, 2012; Carlini and Wagner, 2014; Davi et al., 2014b; Göktas et al., 2014; Göktaş et al., 2014; Bittau et al., 2014; Snow et al., 2013; Offensive Security, 2014; Kemerlis et al., 2012, 2014).

To address limitations of existing kernel CFI systems, and thereby enhance existing forward control-flow CFI techniques so that they provide a complete defense, we present SecRet, the first microarchitectural context-sensitive full system return integrity mechanism, hardware or software, that mitigates return based operating system exploits by enforcing kernel-return integrity (KRI). SecRet hardware dynamically records *return-to-sender* control-flow targets to an on-chip secure return address stack buffer called the SecRet buffer and supports multi-threading with the first hardware mechanism that detects thread creation and context-switching. SecRet Stacks are managed by a small firmware component that pages inactive stacks to a hardware-isolated subset of main memory. The effect of our approach is the complete removal of software modifiability of return addresses, which effectively enforces operating system "return-to-sender" semantics.

While previous secure return address approaches (McGregor et al., 2003; Lee et al., 2003; Ozdoganoglu et al., 2006; Xu et al., 2002; Kayaalp et al., 2012) are context-sensitive and in-hardware, they only support user level function return integrity (which is also the case for compiler split stack approach SafeStack (Kuznetsov et al., 2014)), and are thus susceptible to operating system memory corruption exploits. In contrast, SecRet introduces several microarchitectural techniques that bridge the semantic gap between hardware and software to transparently manage all protected stack operations: synchronize SecRet Stacks with their corresponding software stacks by detecting thread creation and context switching, isolate paged return data from operating system memory access, and detect and record dynamic return locations for system level events (*e.g.,* interrupts).

Moreover, we evaluate the feasibility of SecRet by prototyping the design in an FPGA (OR1200 (Lampret et al., 2014)): the first full-system real-hardware implementation of any hardware-assisted shadow stack design, that we know of. The FPGA prototype ensures KRI on both function- and system-level return types. SecRet, with a hardware buffer of 8 entries, adds approximately 6% area and less than 1% performance overheads and boots Linux with only two minor changes (which would be unnecessary with the implementation of SecRet software exception handling instructions as presented Section 9.4.5), while supporting 20 separate SecRet Stacks with an average nested depth of 12.

We also address key, but previously unexplored, compatibility issues required for any hardware-assisted shadow stack to be applicable to operating systems and deployable in practice. Specifically, SecRet considers returns under exceptional control-flows: signal delivery, `longjmp`, and `try/catch` (*e.g.,* C++ exceptions), none of which are properly handled by previous work. First, SecRet presents new hardware detectors to transparently detect and manage operating system signal delivery and return. **Second, we identify that several existing hardware-assisted shadow stack approaches do not correctly address `setjmp/longjmp` or software exception handling, and that the primary approach to solving `setjmp/longjmp`, called long distance returns, may lead to unique security violations not present in coarse-grain static CFI.**

We address these issues by extending SecRet with four new software exception handling instructions that accomplish two tasks: 1) properly synchronize the SecRet Stack on `longjmp` and `try/catch` exceptions and 2) enforce software exception handling CFI—thereby defending a common and powerful attack vector (Checkoway et al., 2010). We implement an x86-64 emulator (Pintool (Luk et al., 2005)) of SecRet and the software exception handling instructions, including compiler modifications to emit one of the instructions, and evaluate compatibility with the LLVM (Lattner and Adve, 2004) *test-suite-3.4* (LLVM, 2014), passing 460/461 compiling tests. This evaluation not only demonstrates the efficacy of our software exception handling design, but also presents a much deeper compatibility study for hardware-assisted shadow stacks in general, demonstrating diverse findings such as compatibility with interpreters like Lua and Python.

Finally, we investigate SecRet's architectural compatibility and security protections. Specifically, in addition to the FPGA and Pintool prototypes we implement an ARM-M0+ simulation of SecRet Stack operations (excluding software exception handling support). We demonstrate user level security by evaluating SecRet against the RIPE attack benchmark (Wilander et al., 2011)—a framework for evaluating user level CFI protection mechanisms, and defend against all return based overwriting attacks.

Overall, our prototypes explore the boundary of the necessary support to completely remove return abstractions from software. These new mechanisms not only protect operating systems from traditional control-flow hijacking attacks, but also protect applications from an attacker using the operating system or any higher privileged software as a conduit to violate return control-flow integrity.

In summary our contributions include:

- The first microarchitectural context-sensitive KRI mechanism, SecRet, that protects user and kernel return addresses from operating system memory corruption attacks while bridging the semantic gap.
- The first that we know of full FPGA implementation of any hardware-assisted shadow stack system, and evaluate its compatibility, area overheads, and performance.
- ISA extensions and compiler prototypes that securely integrate software exception handling with context-sensitive function returns, which also enforces `setjmp/longjmp` and presents the first such solution for `try/catch` CFI.
- We demonstrate the compatibility of SecRet design with two diverse architectures, OR1200 FPGA and an ARM-M0+ simulator.
- We evaluate for the first time compatibility of the context-sensitive return policy and SecRet software exception handling instructions with a large corpus of existing software including Linux boot, scripting languages, and successful execution of 460/461 unique benchmarks from the LLVM test suite infrastructure.

9.2 Background and Motivation

In general, the goal of an attacker is to coerce the attacked system into unindented behavior, with the most powerful attacks allowing full attacker controlled execution with the authority of the exploited process: control-flow hijacking attacks. In this section we detail key elements of control-flow hijack attacks, present the security policy, kernel-return integrity, that SecRet uses to mitigate return based attacks, and describe related state-of-the-art mitigations.

9.2.1 Why Full System Return Integrity?

Control-flow hijack attacks operate by exploiting memory corruption vulnerabilities to either modify existing code in place and/or modify code-pointers to target attacker controlled operations. Return control flows are a fundamental element of ensuring control flow integrity. Return addresses are critical to protect and often targeted by attackers because 1) all instances are indirect in contrast to forward control-flow transfers even if static analysis can determine some are direct, 2) they occur with high frequency not only at runtime but also in the static code, 3) return addresses are typically in well known locations, *i.e.,* stack memory region, and 4) return addresses enable ROP.[2] Consequently, function returns are consistently employed by control-flow hijack attacks, being used as both an entry point to divert control flow (One, 1996; van der Veen et al., 2012; Szekeres et al., 2013) and as the enabling mechanism for return-oriented programs for both user level and operating system exploits (Shacham, 2007; Roemer et al., 2012; Carlini and Wagner, 2014; Davi et al., 2014b; Göktas et al., 2014; Göktaş et al., 2014).

Not only is kernel-return integrity a significant and challenging property for operating systems, substantial evidence supports the need for composing context-sensitive function returns with other mitigations to achieve effective defenses against all control-flow hijack attacks. We know from security literature that in general, systematic defenses to kernel and user level control-flow hijack attacks require a combination of protections: code execution integrity ($W \oplus X$)—to defend against code injection; integrity of forward control-flows—*e.g.,*`jmp`, `call`, signal delivery, `longjmp`; and return control-flows—traditional function level returns as well as system level returns like returns-from-interrupts. Return integrity is one of the most challenging because it requires the full isolation and protection of return data, but that data is modified frequently. Several existing popular approaches depend upon efficient support of context-sensitive function return including: the original CFI work by Abadi et al., CPI and SafeStack, KCoFI, and many more. Furthermore, it is typical for CFI solutions to assume $W \oplus X$, for example the original CFI and CPI do so, and we contend that the same is true of breaking CFI into the constituent forward control-flow and return control-flow mitigations. Lastly, we argue that the integration of context-sensitive function return approaches, as

[2] We distinguish return-oriented programming (ROP) from jump-oriented programming (JOP) and always specifically use ROP to denote methods using the return-based mechanisms to execute arbitrary code.

Control-Flow Type	**Policy**
return-from-function	Context-sensitive return
return-from-interrupt	Context-sensitive return
return-from-signal	Record valid thread return location prior to signal
return-from-context-switch	Ensure proper SecRet Stack on context-switches
return-from-`longjmp` (SEH)	Return to any `setjmp` frame on the SecRet Stack while synchronizing SecRet Stack
return-from-`unwind` (SEH)	Return to closest nested `catch` block matching the SEH landing pad while synchronizing SecRet Stack

Table 9.1: Kernel Return Integrity: types of return path protections enforced.

exemplified by both CFI and CPI, with forward control-flow mitigations suggests the compatibility and suitability of combining SecRet with alternative forward control-flow CFI techniques.

9.2.2 Microarchitectural Kernel Return Integrity

SecRet enforces the kernel-return integrity (KRI) policy, which provides control-flow integrity for a subset of system level indirect control flow transfers: protect all return path operations as presented in Table 9.1. Specifically, KRI starts with the model of protecting straight line, uninterrupted program execution such that all function calls return-to-sender, which is the traditional context-sensitive return control-flow policy as enforced by shadow stacks (Abadi et al., 2009). However, in SecRet this policy is enforced for the **entire system**, which differs from those enforced by related hardware-assisted shadow stack approaches(McGregor et al., 2003; Lee et al., 2003; Ozdoganoglu et al., 2006; Xu et al., 2002; Kayaalp et al., 2012). Additionally, although KCoFI (Criswell et al., 2014a) and HyperSafe (Wang and Jiang, 2010) enforce CFI for operating system function returns, SecRet provides context-sensitive protections instead of the static CFI policy, and furthermore, SecRet transparently protects all software return integrity, including for all applications.

Beyond function returns, KRI supports return integrity for system level events, most of which cannot be statically derived because the data is only present at runtime: thus, by definition KRI for kernel level returns is context-sensitive. KCoFI (Criswell et al., 2014a) introduced the particular types of control-flows that must be protected for kernel CFI, including return-from-interrupt and return-from-signal, however, KCoFI manages context switching and therefore by design ensures proper stack selection. In contrast, SecRet protects the same control-flow operations but does so using hardware detectors.

These policies realize full system context-sensitive KRI; however, they will reject exceptional control-flows. Therefore, SecRet also considers policies that enable secure exceptional control flows: `setjmp/longjmp` and `try/catch` software exception handling (SEH)—we refer to both of these as SEH. First we restrict `longjmp` control-flow transfers to target only valid previously inserted `setjmp` stack frames (`setjmp/longjmp` operation is described in Section 9.4.5). Due to the dynamic nature of `setjmp` the most precise policy that we can achieve is to allow control-flow to return to any on stack `setjmp` location, thereby rejecting `longjmp` targets of frames that have been removed from the stack, which is considered an undefined behavior by language specifications (*e.g.,* C90, C99). Second we enforce that `try/catch` SEH, those using `unwind` operations to return to the `catch` block, only targets legitimate previously invoked `try/catch` locations: the specification for `try/catch` exception handling (Samuel, A; LLVM) requires that the closest matching nested `catch` block be targeted thereby restricting exceptions from jumping over a matching `catch` block—note that this may jump over non-matching landing pads.

The last significant aspect of the SEH-CFI policies that we enforce is that they have the added contribution of protecting the forward-flow transfers of their respective CFI targets. Therefore, these policies defend against forward flow attacks using either `setjmp/longjmp` or `try/catch` SEH.

9.2.3 Exploit Mitigations

SecRet prevents kernel control-flow hijack attacks by enforcing kernel-return integrity using complete hardware isolation—eliminating the abstraction of independent returns. In this review we discuss kernel CFI approaches and stack protection approaches. Due to their probabilistic nature we do not consider randomization or time based detection techniques, but readers are encouraged to review recent systematization of knowledge papers on the broader related work (van der Veen et al., 2012; Szekeres et al., 2013).

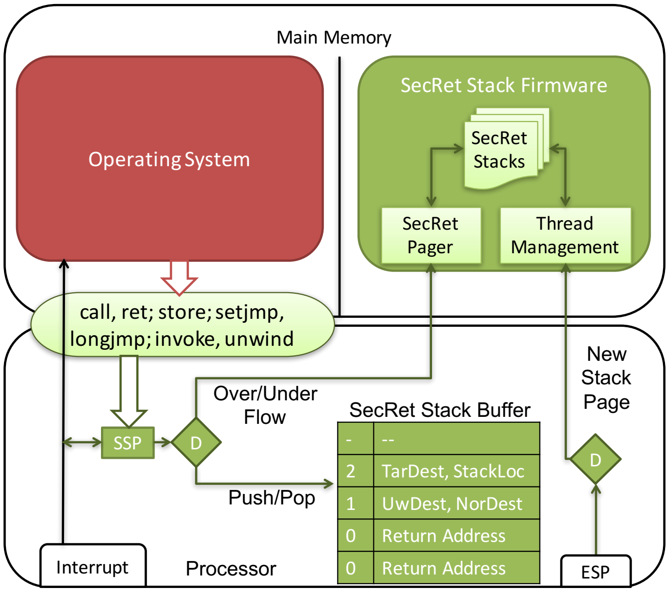
Kernel CFI

Total-CFI (Prakash et al., 2013) enforces a weak variant of kernel-return integrity, that was independently proposed in 2013: it detects new threads and processes to create a shadow stack that enforces context-sensitive function returns, enforces return integrity for signals, enforces forward-flow integrity, and enforces long distance returns for software exception handling. However, Total-CFI, depends upon hard coded information about the kernel to successfully detect and enforce context-sensitive function return integrity for threads, employs imprecise policies for both system level returns (interrupts and signals) and software exception handling, is designed for Qemu making it a virtual machine monitor based solution, and has average overhead between $6 - 19\%$. In contrast, SecRet introduces a complete software transparent thread detector that uses physical pages to match the SecRet Stack with each thread without any operating system information, is implemented in an FPGA and ARM based prototypes, and provides strict CFI for signals and software exception control-flows.

User Mode Function Level Return Integrity In order to protect return integrity for operating systems, return addresses must be made immutable from dynamic modification. There are several approaches that successfully provide these protections for user level code by using either a shadow or split stack; however, absolutely none of the existing mitigations—including software shadow stacks (Cowan et al., 1998; Frantzen and Shuey, 2001; Abadi et al., 2009; Prasad and Chiueh, 2003; Chiueh and Hsu, 2001; Giffin et al., 2002), hardware shadow stacks (McGregor et al., 2003; Lee et al., 2003; Ozdoganoglu et al., 2006; Xu et al., 2002; Kayaalp et al., 2012), or split stack approaches (Kuznetsov et al., 2014)—provide return address integrity for operating system code. The same is true of other hardware based protection mechanisms (Crandall and Chong, 2004; Corliss et al., 2005; Davi et al., 2015); specifically, HAIFIX registers active functions as they are called and restricts returns to call-preceded locations in the set of active functions: a similar policy to long distance returns (Davi et al., 2014a, 2015); in contrast, SecRet enforces *context-sensitive* function returns, is faster (>2x), and also enforces kernel-return integrity. DISE (Corliss et al., 2005), independent from our work, identified that the long distance returns for `longjmp` handling creates incompatibilities, may lead to security compromise, and proposed a similar solution to handling `longjmp` SEH as SecRet; however, SecRet presents examples of these problems, uses new instructions to support SEH CFI as opposed to a prototype dynamic instruction rewriting technique of DISE, and also introduces a solution for `try/catch` SEH. The primary limitation to all these approaches is a lack of memory isolation for return address data while it is off the chip, and that they only address user level function return integrity: in contrast, SecRet introduces novel hardware design and implementation that not only enforces kernel-return integrity but completely removes the abstraction of independent returns from all software, completely redefining the hardware software boundary.

9.3 Threat Model and Assumptions

We assume the following threat model, which is similar to those used in prior work (Carlini et al., 2015; Criswell et al., 2014a): the operating system is not malicious but may contain vulnerabilities, just as user mode software; an attacker that can arbitrarily modify unprivileged processor state (*e.g.,* registers and cache) and memory—including return addresses on the traditional stack, and interrupted program state; the SecRet stack management firmware is free of any vulnerabilities. Additionally, we assume that an attacker may try and violate control-flow through the use of `setjmp/longjmp` or language level exceptions. Note that by itself, SecRet does not defend against code injection, non-control data attacks (Chen et al., 2005) or jump-oriented program (JOP) attacks (Checkoway et al., 2010), we propose to address these concerns in future work (Section 11).



-	--
2	TarDest, StackLoc
1	UwDest, NorDest
0	Return Address
0	Return Address

Figure 9.1: SecRet Architecture

We also do not address availability attacks where an attacker might attempt to exhaust the SecRet resources.

9.4 SecRet Design

The goal of SecRet is to implement a microarchitectural context-sensitive kernel return integrity mechanism: the end result of our design process is the first approach that completely isolates return addresses from the entire software stack, including the operating system, and even CPU buffers (*i.e.*, caches and TLBs). Specifically, SecRet bridges the semantic gap between hardware and software to transparently associate SecRet Stacks to their corresponding software thread and manages all SecRet Stacks in hardware isolated memory. Additionally, SecRet allows legitimate exceptional control-flow behaviors while preserving SecRet Stack integrity.

9.4.1 Design Principles

In order to make SecRet practical, lightweight, and feasible for inclusion into commodity systems we present the following guiding principles:

Minimal CPU Modifications SecRet should require as minimal core CPU changes as possible to both ease the path to adoption and to avoid impacting critical CPU space resources normally used for elements such as on chip caches.

Architecture Independent Many architectures have been shown to be susceptible to ROP attacks (Shacham, 2007); we believe that any hardware-based solution should apply to a diverse set of architectures.

Absolute Dynamic Return Integrity Static policies for managing control-flows continue to be circumventable. Therefore, we select the most precise policy: context-sensitive returns.

Microarchitectural Complete Isolation Software continues to be vulnerable. Therefore, we prioritize full isolation, including from indirect control of operating system, and once that is achieved investigate secure methods to handle exceptional control-flows.

Performance If SecRet's protections are costly in terms of added hardware or increased software run times, no one will use it. Furthermore, given the rise of ROP and the need for context-sensitive returns, hardware presents the most secure and performant option.

Explicit Interfaces for Compatibility By fully isolating SecRet Stacks, we provide rich protection guarantees, but at the cost of compatibility. Therefore, SecRet includes controlled, well defined interfaces to maintain compatibility while enforcing return integrity: a critical feature for practical deployability.

9.4.2 System Overview

SecRet transparently enforces kernel-return integrity by interposing on all control-flows that could either directly or indirectly violate return integrity, including function call returns as well as bi-directional returns between system and user level code. SecRet, therefore, decomposes into single-threaded, non-system interactive functionality, function return integrity, and multi-threaded operations, system return integrity, where SecRet ensures return integrity when dealing with system level events such as context-switches, interrupts, and thread creation. SecRet, as depicted in Figure 9.1, realizes these two sets of functionality by adding two new microarchitecture elements to the CPU, the SecRet buffer and associated SecRet Stack pointer register (SSP in the diagram), and extends existing exception handling hardware infrastructure with signals to handle SecRet Stack related events. SecRet firmware comprises the exception handlers and memory for storing inactive SecRet Stacks. The figure depicts the SecRet Stack with 4 pushed frames including a `setjmp` frame and `invoke` frame: the frame type is indicated by the flag in the entry. The exception handler decodes and delivers all SecRet exceptions to their respective handlers, which includes both single-threaded and multi-threaded exceptions. Finally, we add instruction set architecture (ISA) extensions to enable correct execution of exceptional control-flows: `setjmp/longjmp` and `try/catch` SEH.

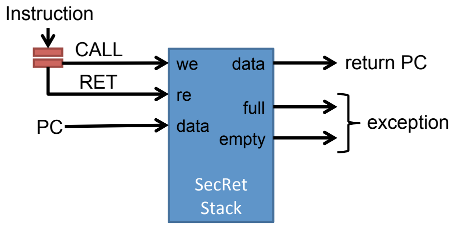


Figure 9.2: SecRet Stack

9.4.3 Function Return Integrity

Automatic SecRet Stack Operation SecRet modifies function call and return instruction logic to automatically interact with the SecRet Stack. Figure 9.2 presents the microarchitecture design for the SecRet Stack. Function calls and returns push/pop return addresses to SecRet buffer as any typical stack, except for when encountering any software exception handling frames on the stack where each SEH frame is popped until the next normal return address. Other than during call operations the SecRet Stack pointer register always points to the top of the stack (least recently pushed return address) or to a reserved stack entry of -1 if the stack is empty. If the address from the SecRet Stack does not match the address from the software's stack then a memory corruption error has occurred: SecRet issues a mismatch exception to be handled by SecRet Stack firmware, which can be silently ignored or since a violation has been detected, to address it accordingly. SecRet leaves policy decisions on how to handle these types of violations to the system designer.

Managing the SecRet Buffer Because hardware is finite, the SecRet buffer may overflow if the nested function call depth exceeds the buffer size. On a full condition, SecRet establishes an overflow signal that invokes the SecRet pager, which copies N items off the bottom of the stack (oldest return addresses) to the top of the paged SecRet Stack in hardware-protected system memory. Then, the firmware moves the remaining hardware stack entries *(Stack Size - N)* down to the bottom of the in-hardware stack and adjusts the stack pointer accordingly. This creates free space at the top of the in-hardware stack, while preserving return address ordering.

Underflows occur when software attempts a return, but the SecRet Stack is empty. In this case, SecRet uses an underflow signal to invoke the SecRet pager. The firmware moves M items from the top of the in-memory stack to the top of the in-hardware stack (preserving order). The firmware finishes by adjusting both in-hardware and in-memory pointers accordingly. If a stack empty condition occurs and there are no swapped-out entries in firmware memory, then SecRet firmware treats this condition as a violation of its security policy: handling depends on if system-level or application-level protections are deployed (SecRet supports both) and is a system designer decision.

Memory Isolation SecRet isolates the SecRet buffer from performance buffers, like the data cache, to eliminate the potential for return address corruption in those general-purpose structures. To accomplish this separation, the SecRet buffer only changes state through calls (pushes an address), returns (pops an address), and when the firmware is running. Right before the processor passes control to SecRet's firmware, it disables both the instruction and data cache, as well as the MMU. These precautions prevent both corruption attacks and control-flow attacks that emanate from general-purpose hardware buffers. SecRet's in-hardware stack is isolated from general-purpose in-hardware buffers, but to prevent return addresses corruption while they are paged out to main memory, SecRet firmware operates in physically-isolated system memory. To achieve this isolation, SecRet includes hardware logic checks that the target address of any load or store instruction comes from an address inside the isolated region of memory. Note that these protections have the added benefit of isolating user level return data from operating system exploits.

9.4.4 System Return Integrity

SecRet must automatically detect particular software state changes to *bridge the semantic gap*. Bridging the semantic gap allows SecRet to avoid depending on the operating system for any information related to tracking new stack creation as well as associating SecRet Stacks with their corresponding software stacks. Otherwise, the operating system could indirectly tamper with return integrity by coercing SecRet into using the wrong SecRet Stack to either launch a complex return-oriented programming attack or corrupt control-flow to launch a denial of service attack.

Thread Creation and Context Switches

In order to detect and handle both thread creation and context switching simultaneously, SecRet adds hardware logic that continuously monitors the stack pointer. Specifically, SecRet looks for when the stack pointer jumps from one *physical* page to another *physical* page[3]. When SecRet detects a cross-page jump, it first saves the in-hardware stack to the in-memory stack for the current thread. Then, it looks to see if the physical page address of the destination stack pointer exists in a list of physical pages of any of the threads that SecRet is already tracking. Each thread has a stack and each stack has an associated list of physical pages that comprise that stack. If the physical page is found to be associated with an existing thread, SecRet sets that thread (and its stack) as active and loads M elements from the in-memory stack into the in-hardware stack. If the physical page is not found, SecRet creates a new thread and adds the physical page as the first item in the list of pages for the newly created thread. SecRet, is able to transparently handle both `fork` and `fork-exec` style process creation events, as well as user level threads that utilize different

[3] An obvious exception to this is when the previous stack pointer address was near the jumped-to page; we consider such transitions to be from the same thread.

physical pages per stack[4].

Interrupt Return Integrity

When an interrupt occurs in the system, either from a system event (e.g., a hardware exception) or synchronous interaction between a user level application and the kernel (e.g., system call) the return address may not be a call-preceded instruction. Therefore, in SecRet we modify interrupt logic to store the program counter to the top of the SecRet buffer so that when the particular interrupted thread is resumed we maintain context-sensitive control flows back to interrupted programs: a similar policy is used by KCoFI to enforce control flow integrity within the kernel (Criswell et al., 2014a); however, in SecRet software never intervenes and maintains the microarchitectural property of enforcing context-sensitive returns. This solution has the added benefit of supporting nested interrupts.

9.4.5 Exceptional Control-Flows

Exceptional control-flows are particular scenarios where the return-to-sender policy may be violated but that are so common place that they must be supported. There are three primary control-flow types that we consider in this work: operating system signal delivery, user application `setjmp/longjmp` and `try/catch`.

Operating System Signal Delivery

In traditional operating systems (*e.g.,* FreeBSD) a signal, *i.e.,* an exceptional system condition, is delivered to the application by 1) adding a signal context onto the user stack, 2) invoking the application signal handler, 3) returning to the kernel's invocation function, and 4) resuming the application by popping off the signal stack frame. SecRet records the return location of the user thread prior to pushing the signal handler context onto the stack, therefore upon return it targets the proper location. The specifics of this depend on the target architecture (Chapter 10). This transparently works in our OR1200 FPGA prototype, but on x86-64 architecture there is a potential problem in that the control-flow transfer to the signal handler in user level code will be different from the SecRet recorded interrupt return location. We propose handling this in future work.

Software Exceptions

As a program executes, particular locations may be marked as "to-be-returned" to under exceptional conditions. These are a form of return-to-sender, but also reflect critical behavior that requires special SecRet Stack synchronization.

[4] We assume that each traditional stack will be unique to a single thread, thus forked stacks will reside on different sets of physical pages from their parents.

To handle these types of scenarios state-of-the-art hardware-assisted shadow stack approaches (Ozdoganoglu et al., 2006; Xu et al., 2002; Kayaalp et al., 2012) propose the long distance return policy. The long distance return policy specifies that returns can target any previous matching hardware-assisted shadow stack entry. Unfortunately, this has three primary limitations: 1) `longjmp` could target legitimate locations causing compatibility problems (a false-positive), 2) allows control-flow to target any return address on the dynamic stack as opposed to only the valid marked target locations, and 3) does not apply to `try/catch` SEH. The second point is the most egregious because if the runtime stack is deep enough it may open up return locations that would not be possible based upon static CFI policies, which is the motivation for a hardware-assisted shadow stack in the first place.

Alternatively, McGregor et al. (2003) proposed a different approach that uses instruction extensions to handle address `setjmp/longjmp` control-flows; however, this work does not consider the architectural design and implementation of synchronizing the hardware-assisted shadow stack on `setjmp/longjmp` events, does not evaluate the design to identify its validity on real workloads, does not consider the effort and changes required to compilers for supporting such policies, and finally does not consider `try/catch` SEH, which requires a different mechanism and recorded stack data to operate securely and correctly. Most importantly, this work assumes that the mere existence of new instructions that allow manual push/pop operations on the hardware-assisted shadow stack solves the problem; however, compile-time information must be encoded into the binary, and it also requires runtime protection of the data that controls the “state-to-return-to”. SecRet addresses the latter problem by adding special SecRet Stack frames to record synchronization points for the SEH control-flows, which is protected from corruption by the firmware.

`setjmp` records the current state of the stack and CPU registers in program memory. This state is passed along a set of function calls to be used by any corresponding `longjmp`. When targeted, `longjmp` restores the stack and CPU state back to the values as passed from `setjmp`. Note that `setjmp/longjmp` represents both a return-to-sender style program flow and a forward flow.

In the benign case the shadow stack will not synchronize with the application stack leading to shadow stack corruption:

```
Calls: A -> F (setjmp) -> B -> A -> D(longjmp) Stack:
A, F, B, A -- lj --> A, F Shadow: A, F, B, A -- lj
--> A, F, B, A
```

The set of calls is listed in the top row, with corresponding stack return locations. Inside of F a `setjmp` is executed, which saves the state of the stack at F. Inside of D a `longjmp` restores the state from F. At that point there is no problem with consistency of the shadow stack because `longjmp` does not do anything to the shadow stack, but when F returns to the first invocation of A, the long distance return policy will target the second frame of A on the shadow stack, which is incorrect.

Handling `setjmp/longjmp`

To correctly and securely handle `setjmp/longjmp`, SecRet introduces two new instructions. The `setjmp` instruction pushes a `setjmp`-stack frame onto the SecRet Stack comprised of: the `setjmp` frame type (integer 2 in Figure 9.1), the main stack pointer location denoting the current top of stack (current depth), and the valid target destination. `setjmp` is intended to be executed as the first instruction of the `setjmp` glibc routine so that all the necessary state is readily available to SecRet hardware (stack pointer holds the address of the stack depth and that value dereferenced is the valid return address).

To `longjmp` to a corresponding location, software manually sets the CPU state, just as the existing glibc `longjmp` implementation does, then executes the `longjmp` instruction, which pops entries off the SecRet Stack until it reaches the matching `setjmp` frame. This approach addresses the compatibility problem of long distance returns because all entries following the invocation to F will be popped off the SecRet Stack. It also addresses the security issue because, although an attacker could modify the destination frame of the preceding `setjmp` data, SecRet forces the jump to target an existing `setjmp` SecRet Stack entry. So in this way SecRet enforces integrity for `setjmp/longjmp` control-flows, a commonly exploited mechanism (Checkoway et al., 2010).

Another potential compatibility problem that could arise is the use of `longjmp`s that target stack frames that are no longer active (*e.g.,* a non-local goto). This is an undefined behavior which permits the compiler to halt execution of the program, which is the solution in SecRet.

`try/catch` SEH

In contrast to `setjmp/longjmp`, `try/catch` have much more strict semantics with respect to valid control-flow: they always return to the closest matching `catch` block (Samuel, A). All exceptions occurring while execution is within the `try` block, including nested function calls, will "unwind" to the corresponding catch block. Exception control flow is well defined under this scenario because every call within the try block has two valid return locations: 1) the normal destination or 2) the *landing pad* location. The *landing pad* roughly corresponds to the catch block. Therefore, SecRet returns to the closest nested *landing pad* on `throw` operations, popping of all other stack frames along the way, or returns as a normal function return.

Unfortunately, `try/catch` has a more complex method for specifying the valid return locations, which requires information only available to the compiler in code generation phases (Samuel, A; LLVM). From SecRet's perspective supporting the `try/catch` requires knowledge about whether or not a particular call instruction occurs within a `try` and where the landing pad is for the `catch`. This information is not readily present to the hardware for automatic recognition, so we add two new instructions with associated compiler support: `invoke` and `unwind`. The `invoke` instruction takes one argument, the landing pad address, and replaces traditional call instructions residing in `try`. When `invoke` is executed SecRet pushes the `invoke` frame onto the SecRet Stack: `invoke` type (integer 2 in Figure 9.1), landing pad destination, and normal return address.

There are two cases for handling returns: 1) normal and 2) `unwind`. In the normal return instance SecRet pops the `invoke` frame and targets the normal return address. Alternatively, an `unwind` requires the stack to be popped until the landing pad destination is at the top of the stack and then transfers control there.

Chapter 10

SecRet Prototypes and Evaluation

To evaluate SecRet we implement three prototypes: OR1200 FPGA that boots Linux, x86-64 emulation in Pintool that shows that SecRet applies to user level x86 binaries, and bare metal ARM Cortex-M0+ simulator that shows that SecRet applies to even the simplest devices. Each prototype in isolation demonstrates specific aspects of our system (e.g., cycle accurate performance evaluations) and in whole demonstrates the architecture independence and software compatibility features of SecRet. Table 10.1 summarizes each prototype with respect to the design elements implemented. Note that the lack of MMU for the ARM prototype does not imply that it cannot achieve holistic protection against memory corruption and code injection attacks because it has a Memory Protection Unit (MPU): the MPU offers access control permissions and therefore can provide $W \oplus X$ properties.

Feature	**x86**	**OR**	**ARM**
CPU SecRet Stack	●	★	★
Call/RET update SecRet Stack	●	★	★
Over/Under flow exceptions	●	★	★
Thread creation detection	●	★	⊗
Context-switch detection	○	★	⊗
Over/Under flow firmware SecRet Stack	○	★	★
SecRet Stack switch on context-switch	●	★	⊗
Interrupt return integrity	○	★	★
Signal handler dispatch	○	★	⊗
C++ exceptions	●	○	○
setjmp/longjmp	●	○	○

Table 10.1: Summary of SecRet features implemented by prototype, organized as hardware support, firmware support, and compatibility. Key: ★ = yes; ○ = no; ● = emulated. ⊗ = not applicable. The ARM platform does not have an MMU, therefore thread related events are not applicable . The x86 prototype is implemented using Pin, the OR (OR1200) prototype is implemented on an FPGA, and the ARM prototype is a cycle accurate instruction set simulator for ARM Cortex-M0+.

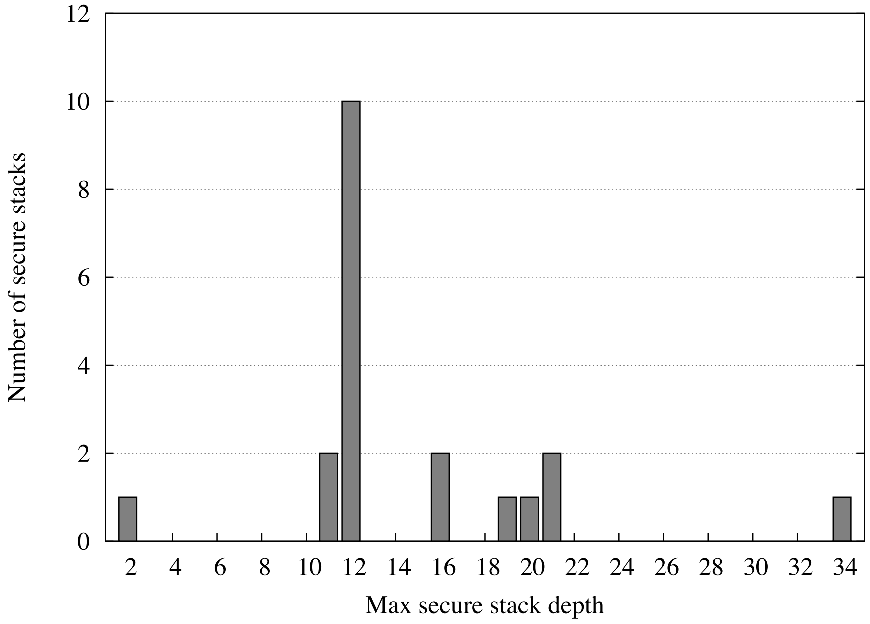


Figure 10.1: Histogram of the maximum stack depth for each of the 20 SecRet stacks required to boot Linux.

10.1 Mobile Prototype

There are two questions that our mobile/FPGA prototype answers: 1) does SecRet support whole-system protection (operating system and user-level programs) and 2) what is the cost of such protections in terms of hardware area. To answer these questions we implement SecRet inside the OR1200 (Lampret et al., 2014)—a processor in the class of mid-range mobile phones. The OR1200 is an open source, 32-bit RISC processor with a 5-stage pipeline, separate data and instruction caches, and MMU support for virtual memory. It is popular as a research prototype and has been used in industry as well. We implement this system on a Xilinx FPGA board and boot a recent version of Linux.

This is the most challenging test of SecRet due to the magnitude and variety of exceptions the Linux kernel uses and, most of all, the fact that there are multiple processes that SecRet needs to manage the SecRet Stacks for. Figure 10.1 shows the results of this experiment: that SecRet supports Linux with one *compiler-provided* modification to support two cases of `setjmp/longjmp` in files sorttexttable.c and recordmcount.c. Once the compiler instruments the code with our new `setjmp/longjmp`-handling instructions, SecRet silently manages all 20 stacks (with a mean maximum depth of 12) that Linux uses during the boot process.

To explore how SecRet Stack size impacts the boot time of Linux, we build six versions of the

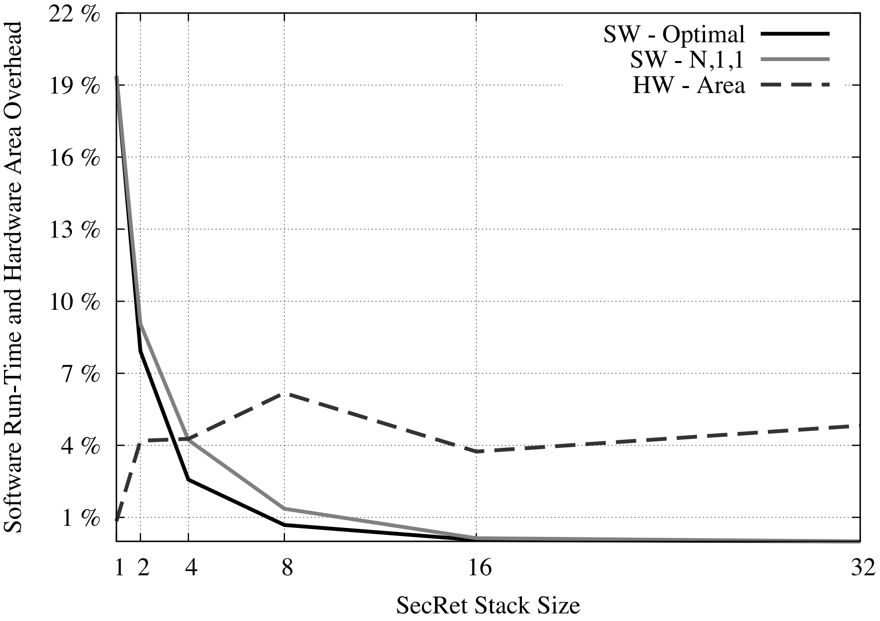


Figure 10.2: The tradeoff space between hardware overhead due to SecRet and software run-time overheads due to firmware stack management for booting Linux.

mobile prototype, each with a different SecRet Stack size. Figure 10.2 shows how stack size not only impacts the hardware area overhead, but also Linux boot time—more stack entries results exponential decay of run-time overhead. Figure 10.2 also shows the effects of two firmware configurations. As Section 10.3 explores, in detail, we represent each SecRet Stack configuration as a three parameter tuple: (number of stack entries, number of entries loaded when empty, and number of entries unloaded when full). For each stack size, N, Figure 10.2 shows both the optimal firmware configuration (*i.e.,* load and unload number) and the $(N, 1, 1)$ configuration that worked best in Section 10.3. A stack size of 16 return addresses leads to less than 4% hardware area overhead and almost 0% performance degradation—which makes sense since the mean max call depth is 12, so the only source of overhead is swapping process stacks.

10.2 x86-64 Prototype

The primary goal of the x86-64 implementation is to 1) investigate the functional correctness and compatibility of the SecRet SEH CFI design and 2) to greatly extend compatibility exploration of previous hardware-assisted shadow stack evaluations: to this end we evaluate SecRet with respect to over 300 unique benchmarks.

10.2.1 Implementation

We implement our x86 prototype using Pin dynamic instrumentation (Luk et al., 2005). For our evaluation we prototype and emulate the features listed in Table 10.1 with the primary purpose of enforcing complete dynamic return integrity. Therefore, we emulate SecRet Stacks by implementing per thread stack data structures in Pin, and handle thread creation by using Pin callback routines. We use Pin's dynamic instruction modification capability to replace all `CALL` and `RET` instructions with routines that emulate these instructions, as well as `setjmp/longjmp` and `invoke` and `unwind`. The SecRet Stack concurrently pushes and pops all three return frame types: normal `CALL` and `RET`s, `setjmp/longjmp`, and `try/catch` SEH frames. One limitation of our implementation is that it does not instrument dynamically linked libraries due to an implementation issue with Pin, but this is not a design limitation. To emulate `try/catch` we add an LLVM compiler pass to emit an intrinsic, `__secret_invoke`, for each `invoke` LLVM intermediate representation instruction, and emulate SecRet hardware using Pin.

10.2.2 Compatibility Evaluation

In an effort to investigate x86 compatibility we evaluate SecRet using the LLVM infrastructure test-suite-3.4 (LLVM, 2014). The LLVM tests that we applied, `TEST=simple`, is comprised of 494 total tests including application, benchmark, and unit/regression tests: we refer the readers to the full list of tests to ascertain the full coverage. For example some of the benchmarks include: BitBench, FreeBench, MallocBench, MiBench, amongst several others. There are also 26 application tests including: Lua, ClamAV, aha, ALAC, lemon, and ldecod+lencod. This test suite is used to perform nightly build test for the LLVM source tree. Therefore, we believe it contains not only a large set of benchmarks and unit tests, but also is representative of the types of programs LLVM expects to work with, as such we believe this to establish a powerful baseline with which to assess SecRet compatibility. Additionally, although the test-suite includes the Lua interpreter we also evaluate Python with pybench (Marc-Andre Lemburg, 2015) to get another sample point.

Our pintool prototype of SecRet successfully passes 460 out of 461 tests that compile, a 99.8% success rate, as well as the Python benchmark. The lone failure is an application level benchmark, SQLite3, and investigation into the cause is ongoing. Thirty-three tests do not statically compile including one from the application suite, Burg, and the rest from benchmark suites including: all PolyBench tests (27/33 tests), Prolangs-C/cdecl/cdecl, Prolangs-C/unix-smail/unix-smail, 7zip/7zip-benchmark, MallocBench/espresso/espresso, and Misc-C++-EH/spirit. It is unclear what cause of the miscompilation are, however, these occur without our LLVM pass and do not indicate failure of the SecRet design.

Exception Handling Analysis The LLVM test-suite has several tests for `setjmp/longjmp` and `try/catch` SEH. Our SecRet prototype successfully completes all of these tests, 13/13 `try/catch` style[1]. We also evaluated six specific `setjmp/longjmp` tests from the 3.5 llvm test-suite that were not in the 3.4 test-suite which includes looping using `setjmp/longjmp` and far jumps. Lastly, several of the application benchmarks, especially the interpreters Python and Lua, that use `setjmp/longjmp` control-flows.

Qualitative Assessment of JIT and Others To ascertain the impact of SecRet on non-C/C++ applications we qualitatively investigated Java SEH and the Firefox SpiderMonkey JavaScript JIT. We found that Java's SEH operates similarly to C++ SEH (Lindholm et al., 2014)—throws return to closest nested catch—and conclude it will be compatible with `invoke` and `unwind` code generation. Next we investigated a popular JIT, Firefox's SpiderMonkey (MDN). SpiderMonkey calls a JIT function, `RunScript` for JITting each function; therefore, it returns from the JIT using `CALL` and `RET` instructions. We conclude that for JITs that operate in this way SecRet will be fully compatible. Despite the small sample size, we believe that JITs in general can accommodate the policies enforced by SecRet. Furthermore, we propose that a `setjmp/longjmp` type solution might work for JITs that manually modify the stack and perform tail-call returns to the JITted code.

10.2.3 Security Evaluation

In this section we evaluate the effectiveness of SecRet by attacking the x86-64 Pintool prototype with the RIPE benchmark suite (Wilander et al., 2011) and two real attacks: code injection via stack overflow and return-oriented program on the stack. The RIPE benchmark (Wilander et al., 2011) is a program that generates a set of vulnerable executables and launches attacks on itself. Although, this is a synthetic benchmark, the attacks will manifest exactly the same way as in the wild exploits and therefore evaluate the efficacy of SecRet. SecRet defends against all 250 attacks using the return address as either the entry point for control flow deviation or as pseudo instruction pointer in code reuse attacks.

Additionally, we deployed two attacks that targeted vulnerabilities in real systems by extracting ROP gadgets from libc. These attacks were obtained from Ben Lynn (Ben Lynn, 2014) and deployed on Linux Mint running a 64-bit 3.8.0-19-generic Linux kernel. The first attack is a traditional buffer overflow that overwrites the return address on the stack and points it at the injected code on the stack. The second attack is a ROP based attack that injects the ROP payload onto the stack while overwriting the return address, then on the first return launches the attack which leads to a shell. SecRet successfully defended against both attacks.

[1] One of these tests launches an exception in the exception handling code, which is in glibc. Our Pintool did not instrument these libraries, so to verify correctness we hardcoded the instruction pointer of the particular nested call instruction to be dynamically rewritten to the SecRet `invoke` instruction and found the test to succeed.

10.3 IoT Prototype

This section covers the challenges and design decisions specific to the Internet-of-Things (IoT) prototype. IoT is defined by connected, low power, and low complexity computational devices. Owing to its class leading low power and extensive tool support, the ARM Cortex-M0+ is one of the most popular processors used in IoT devices. The Cortex-M0+ implements the ARMv6m (Thumb only) instruction set. It has a single issue, in-order, two-stage pipeline with single cycle access to RAM and Flash memories—no cache required. IoT devices tend to be single function devices, thus there is no memory management unit. The processor runs at 24 MHz.

The goal of the experiments on this platform is to show that SecRet scales to simple devices: small SecRet Stack buffers do not hinder the performance of the types of applications that run on processors that necessitate having a small buffer. The simplicity of the IoT platform also allows us to explore more deeply the impact of configuration on overhead and how the empty and full firmware handlers contribute to the overhead of SecRet for a range of applications. Table 10.1 shows the challenges that our IoT prototype addresses.

10.3.1 Details

To implement SecRet on the ARM Cortex-M0+, we hook the PUSH ..., LR, POP ..., PC , and bx LR instructions. We do not need to hook the STM and LDM instructions, since they can only interact with the low eight registers: not the LR or PC. We can ignore MOV LR and ADD LR since targeting the link register is not supported per the specification. Lastly, we ignore BLX reg, since the Cortex-M0+ is a thumb only processor.

10.3.2 Evaluation Setup

To evaluate the effectiveness of SecRet on an IoT platform, we create an cycle-accurate instruction set simulator for the ARM Cortex-M0+. Using a simulator allows us to easily explore how turning the various design knobs of SecRet (*i.e.,* stack size, number of items loaded when empty, and number of items unloaded when full) impacts software run time overhead. Being cycle accurate means that our overheads are exactly as they would appear in hardware. The simulator does occlude the hardware overheads, which we address in our Mobile prototype in Section 10.1.

For benchmarks, we use MiBench (Guthaus et al., 2001). MiBench is a popular embedded systems benchmark that contains six categories of applications: automotive, consumer, network, office, security, and telecomm. Each category is defined by the similar characteristics (*e.g.,* computation bound) of the applications in that category. Thus, we select one application from each category for our evaluation.

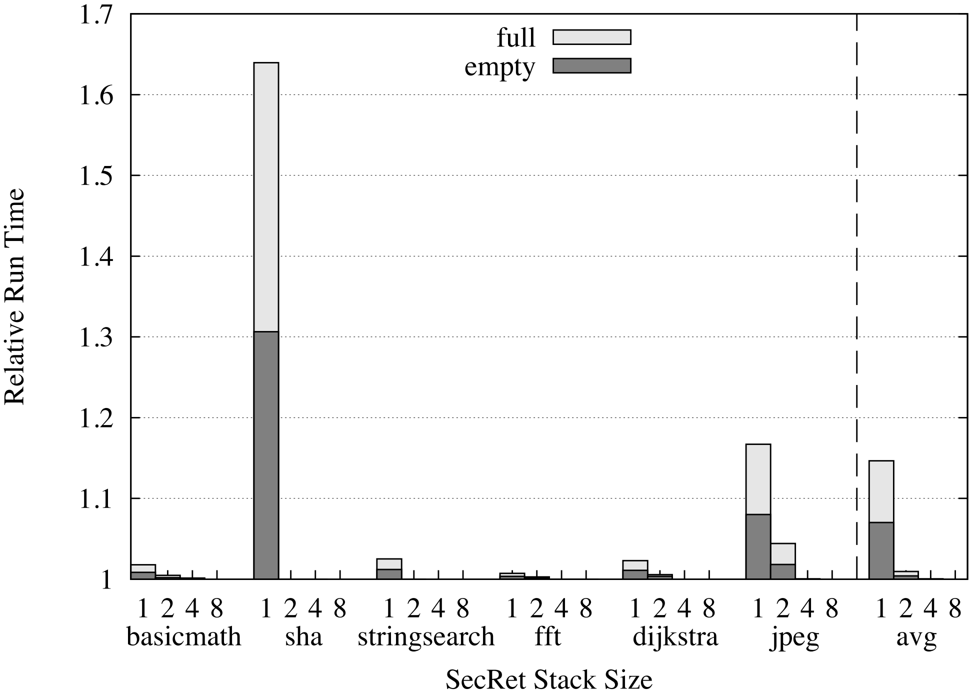


Figure 10.3: Software run-time overhead for each benchmark and the average across all benchmarks, broken-down by source.

10.3.3 Software Overhead

Figure 10.3 shows the software run-time overhead associated with four different SecRet Stack sizes. For this experiment, we choose the optimal firmware configuration (we explore the impact of firmware configuration next). Note that since these devices are single function, it is reasonable to expect software designers to tune the firmware to minimize overhead. Five of the six benchmarks have maximum stack depths between four and eight return addresses deep; the fft benchmark is the outlier with a maximum depth of 22 addresses. This is why a stack size of only four entries leads to essentially zero overhead. Reducing the stack size to two entries leads to only a 1% performance overhead. Also interesting is the *sha* result. The *sha* benchmark has a shallow stack, but has a high frequency of function calls. This leads to the high overhead for the single entry case, but to no overhead once the SecRet Stack supports two return addresses.

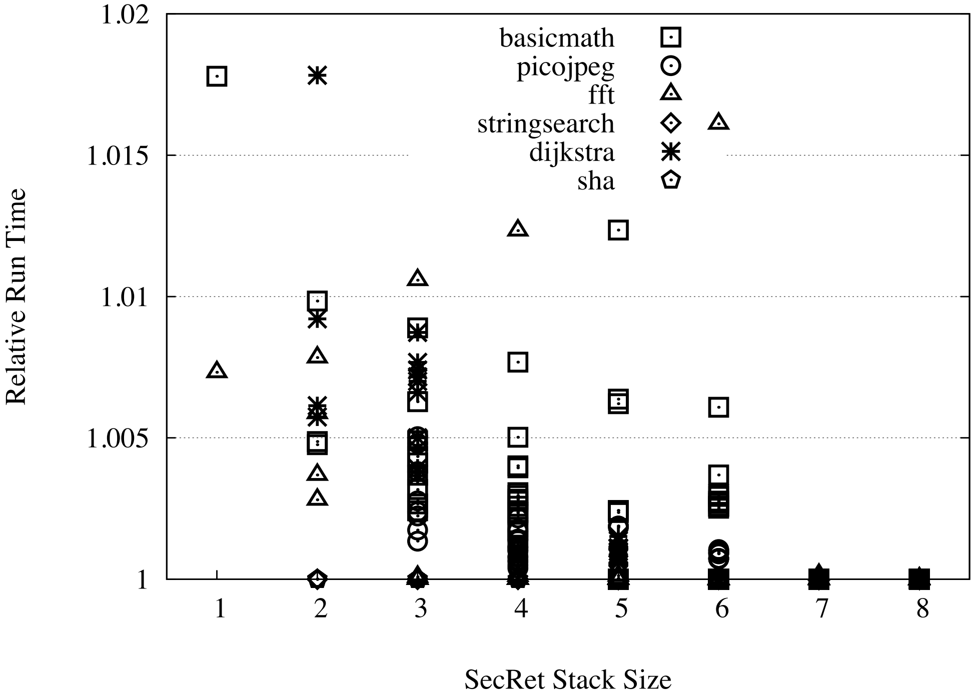


Figure 10.4: Software run-time overhead for different stack size, load size, and unload size for each benchmark. Zoomed-in to show how most configurations approach the optimal run-time overhead.

10.3.4 Configuration Impact on Overhead

Figure 10.4 shows the impact of tuning both the firmware parameters (number of address moved from the firmware stack to the hardware stack in an empty and number of addresses move from the hardware stack to the firmware stack in a full) and hardware parameters (SecRet Stack size). The vast majority of configurations+benchmark combinations result in less than 2% performance degradation. Only four configuration+benchmark combinations—out of 1224—result in greater than 20% performance degradation (not shown due to scaling). The outliers all come from the *sha* benchmark with stack sizes four and less. These results show a trend that when the load and unload size is set to the size of the stack—*i.e.,* swap out/in the full stack on each over/under flow—is generally a pathologically bad performer and single entry load/unload is generally the best or near the best performer. In addition, outside of pathological cases, having the firmware load and unload different number of entries to and from the stack does *not* impact performance greatly.

Chapter 11

SecRet Discussion and Future Work

The goal of SecRet is to investigate the extent to which we can completely remove the software visible abstraction of indirect returns at the ISA level. In so doing we introduced new detectors for thread creation and switching events; exposed several implementation artifacts through our FPGA prototype; and specifically revealed issues in full system deployment: portability, performance, and compatibility. In this chapter we detail key lessons that we learned that were previously unexplored and highlight the key limitations and roadblocks to completely deploying an approach like SecRet.

`try/catch` SEH Requires Compiler Modification We found that although `setjmp/longjmp` SEH can be modified to abide by kernel-return integrity, standard `try/catch` SEH cannot because it depends upon information (*i.e.,* normal and throw based return locations) that is only available to deep machine code generation passes of LLVM. Because of this, the methods proposed by previous secure return address stack research (McGregor et al., 2003) are insufficient to address compatibility of `try/catch`.

Make the Common Case the Common Case There are several control-flow patterns that adhere to the return-to-sender policy; SecRet demonstrated several in the specification and enforcement of kernel-return integrity. We also demonstrated how these common case return-to-sender control-flows can be the *fail-safe default* configuration, while allowing non-conformant exceptions to the common case through an SEH ISA interface and compiler modifications. The benefit is the complete *elimination of all return address overwriting hijacks as well as ROP payload attacks.* Given the rise of ROP and the inability of all software based approaches to fully mitigate its threat, we believe that SecRet and similar hardware based mitigations are not only viable but their necessity is justified.

Supporting Dynamic Memory Isolation Despite the full isolation of SecRet stacks, SecRet does not dynamically allocate or free memory on thread creation and exit events (Prakash et al., 2013). we propose dynamic memory allocation could be supported by the Nested Kernel (Dautenhahn et al., 2015), or even Intel's Software Guard Extensions (SGX) for memory isolation. SGX provides a particularly interesting direction in that it behaves similar to our firmware isolation and integrates with x86-64. Regardless of mechanism employed, dynamic memory requirements for secure return address stacks is a necessary challenge to conquer for SecRet to be realized in practice.

Composition of SecRet with Forward-Flow and Code Integrity For complete CFI protection, we advocate for a defense that employs SecRet return address protections in combination with existing, forward control flow, defenses (Abadi et al., 2009; Kuznetsov et al., 2014). We believe that both forward control-flows and return control-flows are essential and complementary, in addition to the necessary protections for system level return paths. As we proposed in Section 9.2.1 we believe that SecRET can be integrated with forward control-flow protections such as KCoFI, and see this as a key future work including integrating the Nested Kernel protections for off-chip stack protections.

Supporting Legacy: PIC/PIE Some mechanisms legitimately use return addresses to compute various system information, *e.g.,* position-independent code (PIC) utilizes return addresses to determine the location of static variables. SecRet does not modify the traditional stack operations: `CALL` and `RET` instructions still push and pop return addresses to the stack. Therefore, we believe SecRet will not hinder compatibility with any such mechanisms.

Chapter 12

Future Work and Conclusions

This dissertation presented the need for commodity monolithic operating system hardening and compartmentalization and proposed that a fundamental impediment to intra-kernel privilege separation is a lack of efficient separation mechanisms and abstractions. We presented the Nested Kernel Architecture, which isolates a system's MMU while nesting the MMU component in commodity monolithic operating systems. The Nested Kernel employs simple existing commodity features (MMU write-protection configurations with kernel depriveleging and kernel code integrity) to isolate the MMU, thereby establishing a memory management protection domain.

The Nested Kernel Architecture demonstrates that real, efficient privilege separation can be retrofitted *in-situ* in a monolithic operating system. We demonstrated that the two Nested Kernel Architecture components, the nested kernel and the outer kernel, can co-exist in the highest hardware protection level in a common address space without compromising the isolation guarantees of the system. This particular technique virtualizes supervisor privilege, which can be used to further isolate other aspects of CPU, memory, and device state. Overall, we presented a new kernel organization that can provide the bedrock for further investigation in privilege separation of commodity monolithic operating system design.

The nested kernel is also representative of a more minimal layer of separation than even microkernel design. This dissertation demonstrated that separation is possible using only the MMU and binary code properties, in contrast to microkernel design that typically includes memory virtualization, scheduling, and inter-process communication. Additionally, we demonstrated that this separation can be inserted with only minimal impact on performance overheads.

This dissertation also demonstrated that the Nested Kernel can efficiently support useful intra-kernel security protections with write-mediation policies, such as write-once and append-only, which OS developers can use to incorporate new security policies with very low performance overheads. By utilizing the write-protection and write-logging services this dissertation specifically demonstrated powerful security protections: kernel code integrity, system call table integrity, shadow process list integrity and recording, and guaranteed invocation and isolation of security monitor events. More broadly, we expect that the Nested Kernel Architecture can improve operating system security by enabling developers to incorporate richer security principles like complete mediation, least privilege, and least common mechanism, for selected OS functionality. A key direction of future work is to extend the Nested Kernel to support the transparent separation of spatial and thread protection domains.

Lastly, this proposal suggests that the Nested Kernel design is feasible to implement in a commodity operating system on commodity hardware and demonstrates that it drastically reduces the TCB for updating the MMU: a reduction of 232 times the size of stock FreeBSD in our x86-64 Nested Kernel prototype PerspicuOS.

This dissertation also observed that certain protection properties such as *return integrity* are extremely challenging to enforce on commodity operating system code because of the authority of the operating system as well as the need for strong isolation with frequent updates to security critical state. Therefore, we also investigated the necessary elements to completely remove all software modifiability of return control-flows in a microarchitectural kernel return integrity mechanism called SecRet. Our evaluation shows that a wide range of applications, including Linux, are naturally amenable to a full-system, context-sensitive return integrity policy implemented via an in-hardware secure return address stack. The evaluation also demonstrates that secure return address stacks apply to a range of architectures, from simple to complex, and have little performance impact on software.

We believe that this evidence not only validates full-system context-sensitive return integrity but also highlights the practicality and deployability of a hardware-implemented secure return address stack. At a high level, the results make it clear that it is necessary—from a security and performance perspective—to attack the malicious control-flow problem in a divide-and-conquer manner; where in-hardware, full-system secure return address stacks guard the return paths and software CFI techniques guard the forward control flows.

At first glance the two approaches, Nested Kernel and SecRet, may appear to have somewhat contrasting goals: 1) the *addition* of a thin MMU virtualization layer to abstract privilege separation in the kernel and on the other hand 2) the removal of an abstraction, *i.e.,* the notion of an indirect return, to eliminate all ROP and return-integrity based attacks. However, both identify fundamental memory and hardware state that is powerful to enforcing mitigation and resiliency, and despite using diverse techniques, both provide a layer of separation between the operating system and this state. Our work on SecRet found that one primary challenge of a practical deployment of a secure return address stack approach is to address the need for dynamic memory management. Therefore, a key future direction is to compose SecRet with the Nested Kernel to overcome this limitation, while also extending the security properties to support forward control-flow integrity. The end result would be a complete defense in depth of operating system control-flow integrity exploits.

Overall, we believe that powerful security properties and fundamental design enhancements can be retrofitted into monolithic operating systems with feasible effort and minimal performance degradation. We believe that SecRet demonstrates that removing certain abstractions, like indirect returns, greatly enhances the security of the system while maintaining commodity operating system compatibility. Finally, we believe that the Nested Kernel Architecture represents the first step of a micro-evolution of commodity monolithic operating system design towards a new class of operating system organization that serves as the foundation with which to decompose and isolate intra-kernel components.

References

Linux kernel multiple function remote memory corruption vulnerabilities, March 2014. http://www.securityfocus.com/bid/66279.

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009. ISSN 1094-9224. doi: 10.1145/1609956.1609960.

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Annual Technical Conference*, USENIX ATC'10, pages 93–112, Altanta, GA, USA, 1986. USENIX Association.

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing Process Isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-578-9. doi: 10.1145/1178597.1178599.

Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 263–277, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.30.

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Manual, Advancd Micro Devices, 2006.

Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, New York, NY, USA, 2013. ACM.

Apple Computer, Inc. Apple Mac OS X kernel semop local stack-based buffer overflow vulnerability, April 2005. http://www.securityfocus.com/bid/13225.

argp and Karl. Exploiting UMA, FreeBSD's Kernel Memory Allocator. *.:: Phrack Magazine ::.*, 0x0d(0x42), November 2009.

Patroklos Argyroudis. Binding the Daemon FreeBSD Kernel Stack and Heap Exploitation, 2010.

Patroklos Argyroudis and Dimitris Glynos. Protecting the Core: Kernel Exploitation Mitigations. *Black Hat Europe'11*, 2011.

Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

P. Auerswald, L. M. Branscomb, S. Shirk, M. Kleeman, T. M. Porte, and R. N. Ellis. Critical Infrastructure and Control Systems Security Curriculum. Training manual, Department of Homeland Security, 2008.

Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 90–102, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2957-6. doi: 10.1145/2660267.2660350.

Ahmed M. Azab, Kirk Swidowski, Jia Ma Bhutkar, Wenbo Shen, Ruowen Wang, and Peng Ning. SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, February 2016. The Internet Society.

Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, pages 82–91, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4250-8. doi: 10.1109/SRDS.2010.39.

Andrew Baker. When code can kill or cure. *The Economist*, June 2012. ISSN 0013-0613.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945462.

Victor R. Basili and Barry T. Perricone. Software Errors and Complexity: An Empirical Investigation0. *Commun. ACM*, 27(1):42–52, January 1984. ISSN 0001-0782. doi: 10.1145/69605.2085.

Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 267–283, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4.

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6.

D. Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: Mathematical Foundations. Technical report, DTIC Document, 1973.

Ben Lynn. 64-bit Linux Return-Oriented Programming. `https://crypto.stanford.edu/~blynn/rop/`, 2014.

B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224077.

Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 227–242, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.22.

Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, November 2012. ISSN 0734-2071. doi: 10.1145/2382553.2382554.

Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7.

Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., August 2015. USENIX Association.

Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-granularity Software Fault Isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles*, SOSP '09, pages 45–58, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629581.

Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 253–264, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451145.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0245-6. doi: 10.1145/1866307.1866370.

Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 74–83, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. doi: 10.1145/237090.237154.

Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 409–, Washington, DC, USA, 2001. IEEE Computer Society.

Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 73–88, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502042.

Kees Cook. Linux kernel CONFIG_HID local memory corruption vulnerability, August 2013a. http://www.securityfocus.com/bid/62043.

Kees Cook. Linux kernel CVE-2013-2897 heap buffer overflow vulnerability, August 2013b. http://www.securityfocus.com/bid/62044.

Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to Protect Return Addresses from Attack. *SIGARCH Comput. Archit. News*, 33(1):65–72, March 2005. ISSN 0163-5964. doi: 10.1145/1055626.1055636.

Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, pages 5–5, Berkeley, CA, USA, 1998. USENIX Association.

Jedidiah R. Crandall and Frederic T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 221–232, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: 10.1109/MICRO.2004.26.

John Criswell. *Secure virtual architecture: security for commodity software systems*. PhD thesis, University of Illinois at Urbana-Champaign, 2014.

John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 351–366, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294295.

John Criswell, Nicolas Geoffray, and Vikram Adve. Memory Safety for Low-level Software/Hardware Interactions. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 83–100, Berkeley, CA, USA, 2009. USENIX Association.

John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 292–307, Washington, DC, USA, 2014a. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.26.

John Criswell, Nathan Dautenhahn, and Vikram Adve. Virtual Ghost: Protecting Applications from Hostile Operating Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 81–96, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541986.

Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 191–206, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694386.

Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, DAC '14, pages 133:1–133:6, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-2730-5. doi: 10.1145/2593069.2596656.

Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014b. USENIX Association. ISBN 978-1-931971-15-7.

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 *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 74:1–74:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3520-1. doi: 10.1145/2744769.2744847.

Dorothy E. Denning. A Lattice Model of Secure Information Flow. *Commun. ACM*, 19(5):236–243, May 1976. ISSN 0001-0782. doi: 10.1145/360051.360056.

Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why Phishing Works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-372-0. doi: 10.1145/1124772.1124861.

Edsger W. Dijkstra. The Structure of the "THE"-multiprogramming System. *Commun. ACM*, 11 (5):341–346, May 1968. ISSN 0001-0782. doi: 10.1145/363095.363143.

Colin Dixon, Ratul Mahajan, Sharad Agarwal, A. J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An Operating System for the Home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 25–25, Berkeley, CA, USA, 2012. USENIX Association.

D. R. Engler, M. F. Kaashoek, and J. O'Toole,Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224076.

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.

Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5, 2011.

Bryan Ford and Russ Cox. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 293–306, Berkeley, CA, USA, 2008. USENIX Association.

Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

Ivan Fratric. Runtime Prevention of Return-Oriented Programming Attacks, 2012.

Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2003, San Diego, California, USA*. The Internet Society, 2003. ISBN 1-891562-16-9.

Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Detecting Manipulated Remote Call Streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-00-5.

Jonathon T. Giffin, Somesh Jha, and Barton P. Miller. Efficient Context-Sensitive Intrusion Detection. In *NDSS*, 2004.

Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 575–589, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.43.

Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, August 2014. USENIX Association. ISBN 978-1-931971-15-7.

Robert P. Goldberg. Survey of Virtual Machine Research. *Computer*, 7(9):34–45, September 1974. ISSN 0018-9162. doi: 10.1109/MC.1974.6323581.

Michael Golm, Meik Felser, Christian Wawersich, and Jürgen Kleinöder. The JX Operating System. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 45–58, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6.

Robert M. Graham. Protection in an Information Processing Utility. *Commun. ACM*, 11(5): 365–369, May 1968. ISSN 0001-0782. doi: 10.1145/363095.363146.

Georgi Guninski. Linux kernel multiple local vulnerabilities, 2005. http://www.securityfocus.com/bid/11956.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workshop on Workload Characterization*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7803-7315-4. doi: 10.1109/WWC.2001.15.

Per Brinch Hansen. The Nucleus of a Multiprogramming System. *Commun. ACM*, 13(4):238–241, April 1970. ISSN 0001-0782. doi: 10.1145/362258.362278.

Ken Herold. Linux in Medical Devices | Medical, 2011.

Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1. doi: 10.1145/2487726.2488370.

Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 265–278, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451146.

Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.*, 6(3):151–180, August 1998. ISSN 0926-227X.

Nima Honarmand and Josep Torrellas. Replay Debugging: Leveraging Record and Replay for Program Debugging. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 445–456, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4.

Nima Honarmand, Nathan Dautenhahn, Josep Torrellas, Samuel T. King, Gilles Pokam, and Cristiano Pereira. Cyrus: Unintrusive Application-level Record-replay for Replay Parallelism. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 193–206, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi: 10.1145/2451116.2451138.

Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 383–398, Berkeley, CA, USA, 2009. USENIX Association.

Galen Hunt, James Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fähndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, and others. An Overview of the Singularity Project1. 2005.

Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. Manual 325384-051US, Intel, June 2014.

Intel. Intel Kernel Guard Technology. `https://01.org/intel-kgt`, 2015.

Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E. Porter, and Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 605–620, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-4686-0. doi: 10.1109/SP.2014.45.

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch Regulation: Low-overhead Protection from Code Reuse Attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 94–105, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2.

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 957–972, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7.

Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945467.

Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-931971-00-5.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596.

Joseph Kong. *Designing BSD Rootkits*. No Starch Press, San Francisco, CA, USA, 2007. ISBN 1-59327-142-5.

Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4.

Damjan Lampret, Chen-Min Chen, Marko Mlinar, Johan Rydberg, Matan Ziv-Av, Chris Ziomkowski, Greg McGary, Bob Gardner, Rohit Mathur, Maria Bolado, Yann Vernier, Julius Baxter, and Stefan Kristiansson. OpenRISC 1000 architecture manual. Architecture Manual Architecture Version 1.1, OPENCORES.ORG, April 2014.

Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 165–182, New York, NY, USA, 1991. ACM. ISBN 0-89791-447-3. doi: 10.1145/121132.121160.

Butler W. Lampson. On reliable and extendable operating systems. *State of the Art Report, Infotech*, 1, 1971.

Butler W. Lampson. A Note on the Confinement Problem. *Commun. ACM*, 16(10):613–615, October 1973. ISSN 0001-0782. doi: 10.1145/362375.362389.

Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, January 1974. ISSN 0163-5980. doi: 10.1145/775265.775268.

Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 33–48, New York, NY, USA, 1983. ACM. ISBN 0-89791-115-6. doi: 10.1145/800217.806614.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9.

Ruby B. Lee, David K. Karig, John Patrick McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In Dieter Hutter, Günter Müller, Werner Stephan, and Markus Ullmann, editors, *Security in Pervasive Computing, First International Conference, Boppard, Germany, March 12-14, 2003, Revised Papers*, volume 2802 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2003. ISBN 3-540-20887-9. doi: 10.1007/978-3-540-39881-3_21.

Jeff Licquia and Amanda McPherson. A $5 Billion Value: Estimating the Total Development Cost of Linux Foundation's Collaborative Projects. Technical report, Linux Foundation, 2016.

J. Liedtke. On Micro-kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: 10.1145/224056.224075.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification.* Pearson Education, 2014.

Linux Foundation. Automotive Grade Linux. `https://www.automotivelinux.org/`, 2015.

LLVM. Exception Handling in LLVM — LLVM 3.8 documentation. `http://llvm.org/docs/ExceptionHandling.html`.

LLVM. LLVM Testing Infrastructure Guide — LLVM 3.6 documentation. `http://llvm.org/docs/TestingGuide.html`, 2014.

LMH. Month of kernel bugs (MoKB) archive, 2006. http://projects.info-pull.com/mokb/.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065034.

Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software Fault Isolation with API Integrity and Multi-principal Modules. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 115–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043568.

Marc-Andre Lemburg. PyBench README. `http://svn.python.org/projects/python/trunk/Tools/pybench/README`, 2015.

Mashable. Self-driving Car. `http://mashable.com/category/self-driving-car/`, 2015.

John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A processor architecture defense against buffer overflow attacks. In *International Conference on Information Technology: Research and Education, 2003. Proceedings. ITRE2003*, pages 243–250, 2003. doi: 10.1109/ITRE.2003.1270612.

Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2118-1. doi: 10.1145/2487726.2488368.

Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

MDN. SpiderMonkey. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`.

Microsoft. Kernel patch protection: frequently asked questions (Windows Drivers). `http://msdn.microsoft.com/en-us/library/windows/hardware/dn613955(v=vs.85).aspx`, 2007.

Sun Microsystems. Sun solaris sysinfo system call kernel memory reading vulnerability, October 2003. http://www.securityfocus.com/bid/8831.

Pablo Montesinos, Matthew Hicks, Samuel T. King, and Josep Torrellas. Capo: A Software-hardware Interface for Practical Deterministic Multiprocessor Replay. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 73–84, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508254.

Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous System Call Detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, February 2006. ISSN 1094-9224. doi: 10.1145/1127345.1127348.

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, May 2005. ISSN 0164-0925. doi: 10.1145/1065887.1065892.

R. M. Needham. Protection Systems and Protection Implementations. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, AFIPS '72 (Fall, part I), pages 571–578, New York, NY, USA, 1972. ACM. doi: 10.1145/1479992.1480073.

P G Neumann. On Hierarchical Design of Computer Systems for Critical Applications. *IEEE Trans. Softw. Eng.*, 12(9):905–920, September 1986. ISSN 0098-5589.

Offensive Security. Disarming Enhanced Mitigation Experience Toolkit (EMET) v 5.0, September 2014.

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

Elliott I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, USA, 1972. ISBN 0-262-15012-3.

Thomas J. Ostrand and Elaine J. Weyuker. The Distribution of Faults in a Large Industrial Software System. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 55–64, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9. doi: 10.1145/566172.566181.

Hilmi Ozdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Trans. Comput.*, 55(10):1271–1285, October 2006. ISSN 0018-9340. doi: 10.1109/TC.2006.166.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4.

D. L. Parnas and D. P. Siewiorek. Use of the Concept of Transparency in the Design of Hierarchically Structured Systems. *Commun. ACM*, 18(7):401–408, July 1975. ISSN 0001-0782. doi: 10.1145/360881.360913.

Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.24.

Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core.* Syngress Publishing, 2010. ISBN 1-59749-486-0 978-1-59749-486-1.

Gilles Pokam, Klaus Danne, Cristiano Pereira, Rolf Kassa, Tim Kranich, Shiliang Hu, Justin Gottschlich, Nima Honarmand, Nathan Dautenhahn, Samuel T. King, and Josep Torrellas. QuickRec: Prototyping an Intel Architecture Extension for Record and Replay of Multithreaded Programs. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 643–654, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485977.

Gerald J. Popek and Charles S. Kline. A Verifiable Protection System. In *Proceedings of the International Conference on Reliable Software*, pages 294–304, New York, NY, USA, 1975. ACM. doi: 10.1145/800027.808451.

Gerald J. Popek and Charles S. Kline. Issues in Kernel Design. In *Operating Systems, An Advanced Course*, pages 209–227, London, UK, UK, 1978. Springer-Verlag. ISBN 3-540-08755-9.

Aravind Prakash, Heng Yin, and Zhenkai Liang. Enforcing System-wide Control Flow Integrity for Exploit Detection and Diagnosis. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 311–322, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1767-2. doi: 10.1145/2484313.2484352.

Manish Prasad and Tzi-cker Chiueh. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks. In *USENIX Annual Technical Conference, General Track*, pages 211–224, 2003.

Chris Ries. Defeating Windows Personal Firewalls: Filtering Methodologies, Attacks, and Defenses. Technical report, 2005.

Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87402-7. doi: 10.1007/978-3-540-87403-4_1.

Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012. ISSN 1094-9224. doi: 10.1145/2133375.2133377.

Alex Roland. Secrecy, Technology, and War: Greek Fire and the Defense of Byzantium, 678-1204. *Technology and Culture*, 33(4):655–679, October 1992. ISSN 0040-165X. doi: 10.2307/3106585.

J. M. Rushby. Design and Verification of Secure Systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM. ISBN 0-89791-062-1. doi: 10.1145/800216.806586.

Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows internals*. Pearson Education, 2012.

Jerome H. Saltzer. Protection and the Control of Information Sharing in Multics. *Commun. ACM*, 17(7):388–402, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361067.

Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

Samuel, A. C++ ABI for Itanium: Exception Handling. `http://mentorembedded.github.io/cxx-abi/abi-eh.html`.

Tom Saulpaugh and Charles A Mirho. *Inside the JavaOS operating system*. Addison-Wesley Reading, 1999.

Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, March 1972. ISSN 0001-0782. doi: 10.1145/361268.361275.

Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294294.

Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-703-2. doi: 10.1145/1315245.1315313.

Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM. ISBN 1-58113-140-2. doi: 10.1145/319151.319163.

Monirul I. Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. Secure in-VM Monitoring Using Hardware Virtualization. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 477–487, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653720.

Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. *Transparent runtime shadow stack: Protection against malicious return address modifications*. Citeseer, 2008.

Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 574–588, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.45.

Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, February 2016. The Internet Society.

Hannes Frederic Sowa. Linux kernel CVE-2013-4470 multiple local memory corruption vulnerabilities, October 2013. http://www.securityfocus.com/bid/63359.

sqrkkyu. Attacking the Core : Kernel Exploiting Notes. `http://phrack.org/issues/64/6.html`, February 2007.

Paul Starzetz. Linux kernel elf core dump local buffer overflow vulnerability. http://www.securityfocus.com/bid/13589.

Paul Starzetz. Linux kernel IGMP multiple vulnerabilities, 2004. http://www.securityfocus.com/bid/11917.

Paul Starzetz and Wojciech Purczynski. Linux kernel setsockopt MCAST_MSFILTER integer overflow vulnerability, 2004. http://www.securityfocus.com/bid/10179.

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 160–171, New York, NY, USA, 2003. ACM. ISBN 1-58113-733-8. doi: 10.1145/782814.782838.

Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, February 2005. ISSN 0734-2071. doi: 10.1145/1047915.1047919.

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. doi: 10.1109/SP.2013.13.

Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure? *Computer*, 39(5):44–51, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.156.

Alexander Tereshkin. Rootkits: Attacking personal firewalls. In *Proceedings of the Black Hat USA 2006 Conference*, 2006.

Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.163.

Inc. Unified EFI. Unified extensible firmware interface specification: Version 2.2d, November 2010.

Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33337-8. doi: 10.1007/978-3-642-33338-5_5.

Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 927–940, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813673.

Ilja van Sprundel. Linux kernel bluetooth signed buffer index vulnerability. http://www.securityfocus.com/bid/12911.

David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 156–, Washington, DC, USA, 2001. IEEE Computer Society.

David Wagner and Paolo Soto. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 255–264, New York, NY, USA, 2002. ACM. ISBN 1-58113-612-9. doi: 10.1145/586110.586145.

David A. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, University of California, Berkeley, 2000. AAI3002306.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM. ISBN 0-89791-632-8. doi: 10.1145/168619.168635.

Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: a Framework for Virtualization-based Security Systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 347–360, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-22-5.

Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4035-1. doi: 10.1109/SP.2010.30.

Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 545–554, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-894-0. doi: 10.1145/1653662.1653728.

Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *1999 IEEE Symposium on Security and Privacy*, SP '99, pages 133–145, Oakland, California, USA, May 1999. IEEE Computer Society. ISBN 0-7695-0176-1. doi: 10.1109/SECPRI.1999.766910.

David Wentzlaff, Christopher J. Jackson, Patrick Griffin, and Anant Agarwal. Configurable Fine-grain Protection for Multicore Processor Virtualization. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 464–475, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2.

David Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`, 2015. http://www.dwheeler.com/sloccount/.

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002. ISSN 0163-5980. doi: 10.1145/844128.844147.

John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 41–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0672-0. doi: 10.1145/2076732.2076739.

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, pages 31–44, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095814.

Emmett Jethro Witchel. *Mondriaan Memory Protection.* PhD thesis, Massachusetts Institute of Technology, 2004. AAI0806132.

C. Wright. Para-virtualization interfaces, 2006. http://lwn.net/Articles/194340.

W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. ACM*, 17(6):337–345, June 1974. ISSN 0001-0782. doi: 10.1145/355616.364017.

Xi Xiong and Peng Liu. SILVER: Fine-Grained and Transparent Protection Domain Primitives in Commodity OS Kernel. In Salvatore J. Stolfo, Angelos Stavrou, and Charles V. Wright, editors, *Research in Attacks, Intrusions, and Defenses*, number 8145 in Lecture Notes in Computer Science, pages 103–122. Springer Berlin Heidelberg, January 2013. ISBN 978-3-642-41283-7 978-3-642-41284-4.

Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*. Citeseer, 2002.

Min Xu, Xuxian Jiang, Ravi Sandhu, and Xinwen Zhang. Towards a VMM-based Usage Control Framework for OS Kernel Integrity Protection. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies*, SACMAT '07, pages 71–80, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-745-2. doi: 10.1145/1266840.1266852.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, SP '09, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi: 10.1109/SP.2009.25.

Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 225–240, Berkeley, CA, USA, 2008. USENIX Association.

Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4.

Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1.