View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Columbia University Academic Commons

Preventing Code Reuse Attacks On Modern Operating Systems

Marios Pomonis

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2020

© 2020

Marios Pomonis

All Rights Reserved

ABSTRACT

Preventing Code Reuse Attacks On Modern Operating Systems

Marios Pomonis

Modern operating systems are often the target of attacks that exploit vulnerabilities to escalate their privilege level. Recently introduced hardening features prevent attackers from using traditional kernel exploitation methodologies and force them to employ techniques that were originally designed for user space exploitation —such as code reuse— to execute arbitrary code with elevated privileges. In this dissertation, we present novel protection mechanisms that render such methodologies ineffective and improve the security of today's operating systems. Specifically, we present solutions that prevent the leakage and corruption of kernel code pointers without employing entities that execute on super-privileged mode (e.g., hypervisors). The leakage of code pointers is an essential step for the construction of reliable code reuse exploits and their corruption is typically necessary for mounting the attack. More concretely, we present the design and implementation of two systems: kR^X and kSplitStack.

kR^X is a system that diversifies the code layout to thwart attackers from constructing code reuse exploits statically. It also prevents the leakage of return addresses through `XOR`-based encryption or by hiding them among decoys (fake pointers to instructions that trap the kernel when executed). Finally, it couples the above with a self-protection mechanism that prevents attackers from leaking the diversified code layout, either by instrumenting every memory read instruction with range checks on x86-64 systems or by imposing limits through the segmentation unit on x86 systems. Evaluation results show that it imposes small runtime overhead on real-world applications when measured on legacy x86-64 systems (~3.63%) and significantly lower on x86 systems (~1.32%) and newer x86-64 CPUs that provide hardware assistance (~2.32%).

kSplitStack, on the other hand, provides stronger protection against leaks of return addresses and guarantees both their secrecy and their integrity by augmenting the isolation mechanism of kR^X on x86-64 systems. This is achieved through a split stack scheme: functions use an unprotected stack for their local variables but switch to a protected one when pushing or poping return addresses. Moreover, kSplitStack protects the secrecy and integrity of control data (e.g., the value of the instruction pointer) in interrupt contexts by redirecting them to protected stacks, thus thwarting attackers from leaking or corrupting code pointers by inducing interrupts or other hardware events. Finally, the evaluation of kSplitStack shows that it imposes a small runtime overhead, comparable to the one of kR^X, both on legacy x86-64 systems (~3.66%) and on newer CPUs with hardware assistance (~2.50%).

Table of Contents

List of Figures

List of Tables

Acknowledgments

Back when I started the program, I had the impression that a PhD is a personal task that one accomplishes by studying, working and experimenting more or less by themselves. Now, at the end of the road it is clear to me that a PhD is a team sport: without the help and support of a lot of people, it is close to impossible to succeed regardless of one's personal effort.

First and foremost, I would like to thank Vasileios P. Kemerlis. This work would definitely not have been possible without his help, support and patience. I feel really privileged that I got the chance to work with him because of both his approach to research —which ultimately shaped mine— and his character. In addition, his (and Jenny's) help when applying for admission and during the early stages of adjusting to my (then) new life cannot be overstated. Thanks Vasili.

In addition, I would like to thank my Columbia advisors, Roxana Geambasu and Angelos D. Keromytis. Even though the project that I worked with Roxana did not materialize to a publication, it allowed me to get a glimpse on her approach to research. Her dedication to her projects and (most importantly) to her students are admirable. Her support and help at that early stage of my PhD were far more important that I believe she realized. Angelos support all these years was also instrumental in the completion of this work.

Some other very important people throughout these years are people I collaborated with: Michalis Polychronakis, Theofilos Petsios, Kangkook Jee, João Moreira. They helped shape both this dissertation and my approach to science, and I would like to sincerely express my gratitude to them.

Having great people at one's work environment is paramount. I was truly fortunate to be surrounded by a lot of great people at Columbia: Suphannee Sivakorn, George Argyros, Angelika Zavou, George Portokalidis, George Kontaxis, Vasilis Pappas, Vaggelis Atlidakis, Eva Sitaridi, Dimitris Paparas, Orestis Polychroniou, Anthi Orfanou, Andrea Lottarini, Adrian Tang, Jessica Rosa, Giannis Karamanolakis and Petros Sousouris to name a few. I hope you enjoyed our time together as much as I did.

Support from friends cannot be overstated. I am fortunate to have a lot of great friends both in the US and back in Greece: Nikos Papanikolaou, Chris Soldatos, Periklis Gkolias, Evripidis Chrisafis, Nikos Papadakis, Ioanna Tzialla, Konstantina Tzialla, Sofia Bakogianni, Kelly Polemi, Marios Georgiou, Alexandra Dimitriou, Haris Fokas, Giannis Gasparis, George Matikas, Isi Hagouel, Xaris Anastos, John Vournous among others. You probably will never know how important you were to the completion of this work and how important you have been (and still are) to me.

Last but definitely not least, nothing would be possible without my family. My parents Thodoros and Nelly, my brother Nikos, my sister-in-law Vivia and my nieces Tania and Lydia. Words cannot express how important you have been to me and how much I love you.

Chapter 1

Introduction

The abundance and diversity of vulnerabilities in Operating System (OS) kernels [147] have made them the target of privilege escalation attacks for over a decade [36, 180]. Traditionally, attackers targeted user space applications that run with system privileges (e.g., servers) to execute arbitrary code with elevated privileges [161]. However after the adoption of a number of user space defenses [46, 143, 152, 153] in modern OSes, they turned their attention to kernel exploitation since exploiting user space applications became significantly more complex and challenging [7]. This turn was further motivated by the limited number of privileged user space applications due to the enforcement of the least privilege principle [167].

In contrast, when targeting the kernel attackers take advantage of its ubiquitous presence in the system execution, which allows them to interact with it from any running user space application (regardless of its privilege level). Additionally, targeting the kernel allows them to exploit the large attack surface [123] that it exposes due to its complex (low level) code base —which spans in tens of millions lines of code [44]— and execute arbitrary code with elevated privileges. Upon successful exploitation, an attacker has an abundance of options since she can corrupt security sensitive data structures to elevate the privileges of user space processes [117, 196], jailbreak devices [1], escape sandboxes [166], disable security protection features [118] or add new privileged users [73].

Kernel software and user space applications suffer from similar types of vulnerabilities since they are written using similar low-level programming languages. The Linux kernel, in 2017 alone, was diagnosed with race conditions [218, 221, 222], buffer overflows [212, 215, 230, 232], use-after-free bugs [217, 229], integer errors [220, 237], memory corruption and memory disclosure vulnerabilities [216, 219, 228, 231, 238], while similar vulnerabilities were found for different kernel vendors (e.g., Windows [213, 214, 233–236], Apple [223–227]). This trend continued in 2018 with over 750 vulnerabilities diagnosed [147]. Attackers traditionally exploited such vulnerabilities using kernel specific techniques based on the shared virtual memory layout between the kernel and user space processes [109], however a number of recently introduced defenses [39, 109, 154, 156, 178, 200], such as SMEP and SMAP, render these techniques ineffective. As a consequence, new kernel exploits employ techniques that rely on *redirecting the execution flow to arbitrary code locations using indirect branch instructions*. Exploits of this type hijack indirect branches to stitch together a sequence of carefully selected code snippets that (when executed) perform the desired arbitrary computation. Since these code snippets are part of the legitimate kernel code, this technique —known as *code reuse*— is impervious to the aforementioned defenses or the W^X policy [132, 185]. Additionally, kernel-space Address Space Layout Randomization (KASLR) [59], a probabilistic defense that randomizes the base address of the kernel, has been shown to be bypassable [93, 100, 106, 134] therefore it does not reliably protect against this type of attacks. To make matters worse, a more powerful strain of this attack, known as *Just-In-Time (JIT) code reuse* [174] takes advantage of memory disclosure vulnerabilities to dynamically leak the contents or the layout of executable pages, thus completely undermining the effectiveness of KASLR. Finally, code reuse attacks are effective on multiple architectures (as evidenced by their user space variant) such as x86/x86-64 [173], ARM [120] and SPARC [19].

1.1 Hypothesis

The introduction of `ret2usr` defenses [39, 109, 154, 156, 178, 200] has forced attackers to employ code reuse methodologies when constructing kernel exploits [1, 16, 166, 196, 198] which, in turn, are impervious to these defenses as well as to the W^X policy [132, 185]. To this end, we hypothesize that the security of modern OSes can be improved by adopting self-protection mechanisms *specifically tailored to the kernel setting* that minimize the set of code pointers an attacker can tamper with to reliably mount code reuse exploits.

1.2 Thesis Statement

This thesis argues that: (a) hiding code pointers, and (b) randomizing the code layout, coupled with a memory isolation mechanism that guarantees the *secrecy* and the *integrity* of the code and the hidden code pointers, can effectively and efficiently protect OS kernels against code reuse exploits.

1.3 Contributions

1. We present the design and implementation of kR^X: a system that protects commodity OS kernels from code reuse attacks. kR^X is based on two pillars: a component that *randomizes* the code layout and a component that enforces the *execute-only memory* principle.

2. We introduce two novel *return address protection* schemes: (a) return address encryption, and (b) return address decoys. Both schemes prevent attackers from leaking return addresses to *infer* the randomized code layout.

3. We introduce three execute-only memory enforcement mechanisms: (a) `kR^X-SFI`: a software-only scheme; (b) `kR^X-MPX`: a hardware-assisted scheme which employs Intel MPX [102], and (c) `kR^X-SEG`: a hardware-based scheme which relies on the segmentation unit available in legacy systems.

4. We introduce `kR^X-KAS`, a new kernel space layout that facilitates the efficient enforcement of execute-only memory. `kR^X-KAS` flips the kernel memory layout to place all code sections on the top of the address space, therefore effectively creating two disjoint regions: one that contains all kernel data sections and another that contains all kernel code sections.

5. We implement kR^X as: (a) a kernel patch to enforce `kR^X-KAS` and `kR^X-SEG`, (b) a

GCC plugin that enforces `kR^X-SFI` and `kR^X-MPX`, and (c) a GCC plugin that diversifies the code layout and implements the two return address protection schemes. We make all of the kR^X code *publicly* available.

6. We assess the effectiveness of kR^X using a real privilege escalation exploit that employs the code reuse methodology targeting Linux kernels. We also adjust (augment) the exploit capabilities to perform direct and indirect Just-In-Time code reuse methodologies. In all cases, kR^X was able to successfully detect and prevent the respective exploitation attempt.

7. We evaluate the performance of kR^X using a set of macro- and micro-benchmarks. Our system incurs small runtime overhead on x86-64 Linux kernels on real-life applications. The overhead drops to negligible when hardware-based XOM enforcement mechanisms are employed both on x86 and x86-64 Linux kernels. The impact on system call and I/O latency and bandwidth is moderate.

8. We present the design and implementation of kSplitStack: an x86-64 system that augments kR^X to further increase the provided protection against code reuse attacks. kSplitStack offers two distinct advantages compared to kR^X: it is not vulnerable to race conditions vulnerabilities and it is able to protect a class of code pointers that kR^X does not protect, namely code pointers emitted during hardware events.

9. We introduce the kSplitStack region, a region that is protected against arbitrary memory disclosure and/or corruption vulnerabilities. The kSplitStack region is a fundamental block of kSplitStack since it holds the sensitive code pointers that should be protected (i.e., return addresses and hardware events code pointers).

10. We introduce a novel return address protection scheme based on *relocation*. It forces the hardware to emit return addresses inside the split-stack region. This scheme guarantees the integrity and secrecy of the protected return addresses in a race-free manner, therefore it is an improvement compared to the aforementioned protection schemes that are based on encryption and deception.

11. We assess the effectiveness of kSplitStack using the same set of exploits that we

employed in the evaluation of kR^X. Additionally, we evaluate it against race condition vulnerabilities that target return addresses and hardware events code pointers.

12. We evaluate the performance of kSplitStack using a set of macro- and micro-benchmarks. We show that it provides better security guarantees than kR^X with a comparable overhead both on legacy x86-64 systems and on newer CPUs with hardware assistance.

1.4 Dissertation Roadmap

This thesis is structured as follows: Chapter 2 provides background information about kernel exploitation, code reuse attacks, and the design space af shadows stacks as a mitigation against control flow hijacking attempts. Chapters 3 and 4 describe the design, implementation and evaluation (both in terms of performance and of security) of kR^X and kSplitStack respectively. Finally, the thesis concludes in Chapter 5.

Chapter 2

Background and Related Work

2.1 Kernel Exploitation

Traditional kernel exploitation techniques rely on the *shared* memory layout imposed by modern commodity OSes (e.g., Linux, Windows, BSDs). This layout places the kernel and user space applications in different regions of the same address space and relies on hardware assistance to prevent user space code (that executes in non-privileged mode) from accessing the kernel code or data. Specifically, the different execution modes of the CPU (i.e., protection rings) [107, 169] in conjunction with the Memory Management Unit (MMU) ensure that less privileged —user space— code is unable to access memory pages that belong to more privileged entities (e.g., the OS kernel). However, until recently, there was no protection against accesses to user space pages from kernel code, a characteristic that attackers exploited for many years when targeting the kernel. Specifically, kernel code had *full* access to the complete address space to facilitate system calls, such as `read()` and `write()`, which require unrestricted access to user space data pages. Additionally, the privileged CPU modes do not prevent the execution of code that resides in user space code pages.

Taking advantage of the above, local attackers traditionally exploited kernel software using a technique called *return-to-user (ret2usr)* [109]. Attackers that employ this technique, control a user space application and attempt to force the execution flow of the kernel code to controlled user space pages that contain code of their choice to execute it with elevated privileges. To achieve this, attackers typically overwrite kernel *code pointers* such as return addresses [171], function pointers [66, 68, 70, 71, 179], and dispatch tables [63, 69] with addresses that point to code that they injected in the user space portion of the address space, so that the execution flow will be hijacked when these code pointers are dereferenced. If control data corruption is not possible, attackers target pointers to *sensitive data structures* in the kernel data sections (e.g., the heap) and redirect them to fake —user space— copies [64, 65, 67, 72]. Typically, these data structures contain code pointers which, in the fake copies, point to the user space code that the attacker wishes to execute [109].

ret2usr attacks take advantage of the weak separation of the user and kernel part of the address space in most architectures. Due to the popularity of ret2usr attacks, recently a number of defenses were proposed that provide strong isolation between the kernel and the user portion of the address space. In this section we focus on defenses that protect the Linux kernel on the x86/x86-64 architecture, even though other architectures have also recently introduced hardening mechanisms against ret2usr attacks [3, 18].

2.1.1 Hardware-based Defenses

Modern Intel CPUs introduce two hardware features, called Supervisor Memory Execute Protection (SMEP) [82] and Supervisor Memory Access Protection (SMAP) [101] that provide strong user-kernel segregation. Both features take advantage of the `User/Supervisor (U/S)` bit in page table entries, that marks whether a page belongs to the OS kernel or to a user space application. Specifically, while the CPU executes in privileged mode (ring 0), SMEP detects attempts to execute code that resides in pages with the `U/S` bit set (i.e., that belong to user space applications) and in such cases triggers page faults, thus preventing the final step (arbitrary code execution) of ret2usr exploits. SMAP complements the above protection by preventing memory accesses (reads/writes) to pages that belong to user space applications, thus countering the second strain of ret2usr attacks that employs fake copies of sensitive data structures. Note that both SMEP and SMAP are enabled by setting their respective bits in the CR4 control register [1] and do not rely on the address space organi-

[1] `CR4.SMEP` and `CR4.SMAP`

sation imposed by the OS (i.e., the fault is triggered regardless of the linear address of the memory fetch).

2.1.2 Software-based Defenses

The PaX team [151] has developed two kernel hardening features that mitigate ret2usr attacks, KERNEXEC (which prevents control-flow hijacking, similarly to SMEP) and UDEREF (which prevents memory accesses in user space, similarly to SMAP). Both KERNEXEC and UDEREF have different designs for x86 and x86-64 due to architectural differences (specifically due to the lack of enforcement of segmentation limits on x86-64). In x86, the kernel portion of the address space is placed in one contiguous region. UDEREF then uses the segmentation unit to prevent attempts to access memory outside the kernel region (i.e., the user space) result in general protection faults. In a similar manner, KERNEXEC limits the `CS` segment so that instruction fetches from user space addresses are not allowed. In x86-64, UDEREF unmaps the user space address range (upon entering kernel mode) and remaps it in a different (shadow) area, which is mapped as non-executable. As a result, attempts to dereference user space pointers —both data and code pointers— trigger page faults since their respective pages are unmapped. On the other hand, KERNEXEC prevents (only) control-flow hijacking using code instrumentation. Specifically, it makes use of the GCC plugin interface and performs bit masking on function pointers and return addresses to confine the execution to the kernel portion of the address space. kGuard [109] is also using GCC plugins to inject control-flow assertions before every indirect branch to prevent executing code from the user space. Additionally, it randomizes the location of those checks to prevent attackers from bypassing them. Finally, Kernel Page-Table Isolation (KPTI) [43], previously KAISER [92], protects against ret2usr attacks since the user space portion of the address space is marked as non-executable on every user-to-kernel context switch.

2.1.3 Kernel vs User Space Exploitation

Both kernel and user space software suffer from similar types of vulnerabilities since they are written in similar, low-level programming languages [212–219, 221–236, 238]. Before the introduction of the defenses described in Section 2.1, adversaries preferred taking advantage of the weak segregation of the kernel and user space portion of the address space when exploiting kernel vulnerabilities. However, after the strong segregation of the address space portions, they are forced to retrofit user space exploitation methodologies in the kernel setting. Fortunately, simple attacks such as code injection (e.g., through stack smashing [149]) are not an option when exploiting a modern OS kernel: the presence of kernel W^X [132, 185] mitigates such attempts. Instead, they are forced to employ more sophisticated attacks such code reuse that we discuss in Section 2.2. Unfortunately, current OS kernel defenses against such attacks are weaker than their user space counterparts: KASLR [59] offers only 9 bits of entropy (on an x86-64 system) which is significantly less than the 28 bits of entropy provided by user space ASLR.

2.2 Code Reuse Attacks And Defenses

Code reuse exploits rely on code fragments (gadgets) located at *predetermined* memory addresses [23, 25, 55, 58, 86, 173]. Code diversification and randomization techniques (colloquially known as fine-grained ASLR [174]) can thwart code-reuse attacks by perturbing executable code at the function [9, 111], basic block [57, 119, 193], or instruction [96, 150] level, so that the exact location of gadgets becomes *unpredictable* [126].

However, Snow et al. introduced "just-in-time" ROP (JIT-ROP) [174], a technique for bypassing fine-grained ASLR in applications with embedded scripting support. JIT-ROP is a staged attack: first, the attacker abuses a memory disclosure vulnerability to recursively read and disassemble code pages, effectively negating the properties of fine-grained ASLR (i.e., the exact code layout becomes known to the attacker); next, the ROP payload is constructed on-the-fly using gadgets collected during the first step.

Oxymoron [6] was the first protection attempt against JIT-ROP. It relies on (x86) memory segmentation to hide references between code pages, thereby impeding the recursive gadget harvesting phase of JIT-ROP. Along the same vein, XnR [5] and HideM [84] prevent code pages from being read by emulating the decades-old concept of *execute-only memory* (XOM) [34, 182] on contemporary architectures, like x86,[2] which lack native support for

[2] In x86 (both 32- and 64-bit) the execute permission implies read access.

XOM. XnR marks code pages as “Not Present,” resulting into a page fault (#PF) whenever an instruction fetch or data access is attempted on a code page; upon such an event, the OS verifies the source of the fault and temporarily marks the page as present, readable and executable, or terminates execution. HideM leverages the fact that x86 has separate Translation Lookaside Buffers (TLBs) for code (ITLB) and data (DTLB). A HideM-enabled OS kernel deliberately de-synchronizes the ITLB from DTLB, so that the same virtual addresses map to different page frames depending on the TLB consulted. Alas, Davi et al. [54] and Conti et al. [31] showed that Oxymoron, XnR, and HideM can be bypassed using *indirect* JIT-ROP attacks by merely harvesting code pointers from (readable) data pages.

As a response, Crane et al. [48, 49] introduced the concept of *leakage-resilient* diversification, which combines XOM and fine-grained ASLR with an indirection mechanism called code-pointer hiding (CPH). Fine-grained ASLR and XOM foil direct (JIT-)ROP, whereas CPH mitigates indirect JIT-ROP by replacing code pointers in readable memory with pointers to arrays of direct jumps (trampolines) to function entry points and return sites—CPH resembles the Procedure Linkage Table (PLT) [142] used in dynamic linking; trampolines are stored in XOM and cannot leak code layout. Readactor [48] is the first system to incorporate leakage-resilient code diversification. It layers CPH over a fine-grained ASLR scheme that leverages function permutation [9, 111] and instruction randomization [150], and implements XOM using a lightweight hypervisor.[3]

LR^2 [17] is a defense system based on a self-protection mechanism that enforces XOM on the code section. Alas, it is tailored to user programs running on mobile devices and uses bit masking to confine memory reads to the lower half of the process address space. Bit masking is not an attractive solution for the kernel setting; it requires canonical address space layouts, which, in turn, entail extensive changes to the kernel memory allocators (for coping with the imposed alignment constrains) and result in a whopping address space waste (e.g., LR^2 squanders half of the address space). KHide [83] on the other hand, protects the

[3] Readactor’s hypervisor makes use of the Extended Page Tables (EPT) feature [139], available in modern Intel CPUs (Nehalem and later). EPT provides separate read (R), write (W), and execute (X) bits in nested page table entries, thereby allowing the revocation of the read permission from certain pages.

OS kernel against code reuse attacks, using a commodity VMM (KVM), however it does not conceal return addresses, which is important for defending against indirect JIT-ROP attacks [31].

SECRET [202] provides XOM-equivalent protection to COTS binaries, using memory segmentation on x86 and information hiding on x86-64, while NORAX [29] leverages a combination of MMU permission bits to retrofit XOM to ARM binaries. As a result, they are only available to architectures that provide native support for marking memory pages as execute-only. More importantly they rely on information hiding to guard against direct JIT-ROP attacks, a strategy that has been shown to be ineffective in the kernel setting [93, 99, 106, 134]. Lastly, they do not perform code diversification thus they do not protect against any kind of attack that relies on pre-computed gadget addresses.

Live Re-randomization Giuffrida et al. [85] introduced modifications to MINIX so that the system can be re-randomized periodically, at runtime. This approach is best suited for microkernels, and not kernels with a monolithic design, while it incurs a significant runtime overhead for short re-randomization intervals. TASR [11] re-randomizes processes each time they perform I/O operations. However, it requires kernel support for protecting the necessary bookkeeping information, and manually annotating assembly code, which is heavily used in kernel context. Shuffler [195] and CodeArmor [28] re-randomize userland applications continuously, treating the OS kernel as part of their TCB. Lastly, RuntimeASLR [135] re-randomizes the address space of service worker processes to prevent clone-probing attacks; such attacks are not applicable to kernel settings.

Other Kernel Defenses KCoFI [50] augments FreeBSD with support for coarse-grained CFI, whereas Fine-CFI [129] and the system presented by Ge et al. [79] rectify the enforcement approach of HyperSafe [192] to implement a fine-grained CFI scheme for the kernels of Linux and FreeBSD, and MINIX and FreeBSD, respectively. In addition, Fine-CFI further improves the enforcement accuracy of Ge et al. by using points-to analysis to obtain a more restricted set of possible targets for function pointers. In the same vein, PaX's RAP [157] provides a fine-grained CFI solution for the Linux kernel. However, though CFI schemes make the construction of ROP code challenging, they can be bypassed by confining the hijacked control flow to valid execution paths [22, 55, 61, 86].

Heisenbyte [181] and NEAR [194] employ destructive code reads to thwart attacks that rely on code disclosures (e.g., JIT-ROP). Alas, Snow et al. [175] demonstrated that destructive code reads can be undermined with code inference attacks. More recently, Pewny et al. [160] further showed that inference attacks can employ whole-function reuse methodologies to bypass destructive code read-based protections, regardless of the underlying randomization. They also propose profiling the program to identify code and data, in an attempt to minimize the code available for disclosure. Similarly to Heisenbyte and NEAR, their system relies on a thin hypervisor that maps code as execute-only.

Li et al. [130] designed a system that renders ROP payloads unusable by eliminating return instructions and opcodes from kernel code. Unfortunately, this protection can be bypassed by using gadgets ending with different types of indirect branches [25, 86]. Chen et al. [26] proposed PrivWatcher, a system that preserves the integrity of process credentials, by placing them in read-only regions and employing a lightweight hypervisor to update them when necessary. Song et al. proposed KENALI [176] to defend against data-only attacks. KENALI enforces kernel data flow integrity [24] by categorizing data in distinguishing regions (i.e., sets of data that can be used to influence access control); its imposed runtime overhead is, however, very high (e.g., 100%–313% on LMBench). Finally, Li et al. [131] note that zero-day vulnerabilities are significantly more common in code paths that are not "popular" (i.e., exercised frequently). With this motivation, they propose Lind, a system that re-creates complex OS functionality using only popular paths; similarly to KENALI, the overhead of Lind is also very high (up to 525%).

2.3 Shadow Stacks

The concept of shadow stack was introduced almost two decades ago [189]. A shadow stack is a safe virtual memory region that holds copies of the real return addresses to ensure their integrity. Typically, when a function is called the return address is copied from the program stack to the shadow stack. This copy is then compared with the address in the program stack when a function exits to detect corruption attempts. There are two major categories in terms of how a shadow stack is designed: parallel [51] or compact [21]. Implementations of the former category [51], place the shadow stack at a fixed offset from the program stack. This design facilitates the quick mapping of return addresses and their corresponding shadow stack copies at the expense of doubling the stack memory size. Implementations of the latter category [30, 45, 56] employ a separate shadow stack pointer to hold the position of the return address copies (e.g., in a register), while the location of the shadow stack is not dependant on the location of the program stack. Even though maintaining a shadow stack pointer incurs performance overhead, it requires less memory since the return address copies are placed sequentially.

Another important design decision that affects the performance and effectiveness of a shadow stack is the method that ensures the integrity of the copies. Solutions that rely on hardware assistance [74, 144, 177] delegate this task to hardware features, while recently Intel Control Enforcement Technology (CET) [103] was announced as an upcoming hardware feature dedicated to providing hardware shadow stack support. Software only solutions on the other hand employ either information hiding techniques to prevent attackers from finding the address of the shadow stack [124, 136] or inline checks and bit masking [124] such as Software Fault Isolation (SFI) [190] to ensure its integrity. Unfortunately, recent attacks [62, 87, 88, 148] have undermined the security of information hiding, while even the most optimized implementations of the latter impose non-negligible performance penalty [172]. Finally, the recently introduced Intel Memory Protection Extensions (MPX) [41] hardware feature can provide hardware-assisted SFI with minimal performance overhead [21].

Chapter 3

kR^X

3.1 Overview

We present `kR^X`: a *comprehensive* and *practical* kernel hardening solution that diversifies the kernel's code and prevents any memory read accesses to it. More importantly, the latter is achieved by following a *self-protection* approach that relies on code instrumentation to apply checks inspired by SFI for preventing memory reads from code sections. *Comprehensive protection* against kernel-level JIT-ROP attacks is achieved by coupling execute-only memory with: i) extensive code diversification, which leverages function and basic block reordering [111, 193], to thwart the direct use of pre-selected gadgets; and ii) return address protection using either a XOR-based encryption scheme [17, 157, 195] or decoy return addresses, to thwart gadget inference through saved return addresses on the kernel stacks [31]. *Practical applicability* to existing systems is ensured given that kR^X: i) does not rely on more privileged entities (e.g., a hypervisor [48, 83]) than the kernel itself; ii) is readily applicable on x86 systems (both 32- and 64-bit), and can leverage support for memory segmentation or protection (i.e., Intel's MPX [102]) to optimize performance; iii) has been implemented as a set of compiler plugins for the widely-used GCC compiler, and has been extensively tested on recent Linux distributions; and iv) incurs a low runtime overhead (in its full protection mode) of 4.04% on the Phoronix Test Suite, which drops to 2.32% when MPX is available, and 1.32% when memory segmentation is in use.

3.2 Threat Model

Adversarial Capabilities We assume *unprivileged* local attackers (i.e., with the ability to execute, or control the execution of, user programs on the OS) who seek to execute arbitrary code with elevated privileges by exploiting kernel-memory corruption bugs [165, 208, 209]. Attackers may overwrite kernel code pointers (e.g., function pointers, dispatch tables, return addresses) with *arbitrary* values [71, 179], through the interaction with the OS via buggy kernel interfaces. Examples include generic pseudo-filesystems (`procfs`, `debugfs` [37, 112]), the system call layer, and virtual device files (`devfs` [122]). Code pointers can be corrupted directly [71] or controlled indirectly (e.g., by first overwriting a pointer to a data structure that contains control data and subsequently tampering with its contents [72], in a manner similar to `vtable` pointer hijacking [168, 183]). Attackers may control *any* number of code pointers and trigger the kernel to dereference them on demand. (Note that this is *not* equivalent to an "arbitrary write" primitive.) Finally, we presume that the attackers are armed with an *arbitrary memory disclosure* bug [204, 207]. In particular, they may trigger the respective vulnerability *multiple* times, forcing the kernel to leak the contents of *any* kernel-space memory address. Microarchitectural attacks, like Meltdown [134], Spectre [114], and similar side-channel attacks [91], are considered out of scope.

Hardening Assumptions We assume an OS that implements the W^X policy [125, 132, 185] in kernel space.[1] Hence, direct (shell)code injection in kernel memory is not attainable. Moreover, we presume that the kernel is hardened against ret2usr attacks. Specifically, in newer platforms, we assume the availability of SMEP (Intel CPUs) [200], whereas for legacy systems we assume protection by KERNEXEC (PaX) [156] or kGuard [109]. In addition, we assume sane (read-only) memory permissions for the Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT) [32, 76]. Finally, the kernel may have support for kernel-space ASLR [59], stack-smashing protection [186], proper `.rodata` sections (constification of critical data structures) [185], pointer (symbol) hiding [164], SMAP/UDEREF [39, 155], page-table isolation (KPTI) [43, 92], or any other hardening feature. kR^X does not require

[1] In Linux, kernel-space W^X can be enabled by asserting the (unintuitive) `DEBUG_RODATA` and `DEBUG_SET_MODULE_RONX` configuration options.

or preclude any such features—they are orthogonal to our scheme(s). Data-only attacks, such as page table tampering [127] or process credentials modification [198], are considered out of scope; (self-)protecting such sensitive data structures [26, 52, 53] is also orthogonal to kR^X.

3.3 Approach

Based on our hardening assumptions, kernel execution can no longer be redirected to code injected in kernel space or hosted in user space. Attackers will have to therefore "compile" their shellcode by stitching together gadgets from the executable sections of the kernel [1, 16, 165, 196, 198] in a ROP [98, 173] or JOP [25] fashion, or use other similar code reuse techniques [23, 55, 58, 86, 187], including (in)direct JIT-ROP [31, 54, 174]. kR^X complements the work on user space leakage-resilient code diversification [17, 48] by providing a solution against code reuse for the *kernel setting.* The goal of kR^X is to aid commodity OS kernels combat: (a) ROP/JOP and similar code reuse attacks [55, 58, 86], (b) direct JIT-ROP, and (c) indirect JIT-ROP. To achieve that, it builds upon two main pillars: (i) the R^X policy, and (ii) fine-grained KASLR.

3.3.1 R^X.

The R^X memory policy imposes the following property: memory can be *either* readable or executable. Hence, by enforcing R^X on diversified kernel code, kR^X prevents direct JIT-ROP attacks. Systems that enforce a comparable memory access policy (e.g., Readactor [48], HideM [84], XnR [5]) typically do so through a *hierarchically-privileged* approach. In particular, the OS kernel or a hypervisor (high-privileged code) provides the XOM capabilities in processes executing in user mode (low-privileged code)—using memory virtualization features (e.g., EPT; Readactor and KHide [83]) or paging nuances (e.g., `#PF`; XnR, TLB de-synchronization; HideM). kR^X, in antithesis, enforces R^X without depending on a hypervisor or any other more privileged component than the OS kernel. This *self-protection* approach has increased security and performance benefits.

Virtualization-based (hierarchically-privileged) kernel protection schemes can be either retrofitted into commodity VMM stacks [83, 129, 158, 163] or implemented using special-purpose hypervisors [48, 181, 191, 194]. The latter result in a smaller trusted computing base (TCB), but they typically require *nesting* hypervisors to attain comprehensive protection. Note that nesting occurs naturally in cloud settings, where contemporary (infrastructure) VMMs are in place, and offbeat security features, like XOM, are enforced on selected applications by custom, ancillary hypervisors [48]. Unfortunately, nested virtualization cripples scalability, as each nesting level results in ∼6–8% of runtime overhead [8], excluding the additional overhead of the deployed protections.

The former approach is not impeccable either. Offloading security features (e.g., code integrity [163], XOM [83], data integrity [191]) to commodity VMMs leads to a flat increase of the virtualization overhead (i.e., "blanket approach;" no targeted or agile hardening), and an even larger TCB, which, in turn, necessitates the deployment of hypervisor protection mechanisms [192, 201], some of which are implemented in super-privileged CPU modes [4, 201]. Considering the above, and the fact that hypervisor exploits are becoming an indispensable part of the attackers' arsenal [80], we investigate a previously unexplored point in the design space.

More specifically, our proposed self-protection approach to R^X enforcement: (a) does not require VMMs [83] or software executing in super-privileged CPU modes [4]; (b) avoids (nesting) virtualization overheads; and (c) is in par with recent industry efforts [33]. Lastly, kR^X enables R^X capabilities even in systems that lack support for hardware-assisted virtualization.

3.3.2 Fine-grained KASLR.

The cornerstone of kR^X is a set of code diversification techniques specifically *tailored* to the kernel setting, to which we collectively refer to as fine-grained KASLR. With R^X ensuring the secrecy of kernel code, fine-grained KASLR provides protection against (in)direct ROP/JOP and alike code-reuse attacks.

In principle, kR^X may employ any leakage-resilient code diversification scheme to defend against (in)direct (JIT-)ROP/JOP. Unfortunately, none of the previously-proposed schemes (e.g., CPH; Readactor [48]) is geared towards the kernel setting. CPH was designed with support for C++, dynamic linking, and just-in-time (JIT) compilation in mind. In contrast, commodity OSes: (a) do not support C++ in kernel mode, hence `vtable` and exception handling, and COOP [170] attacks, are not relevant in this setting; (b) although they do support loadable modules, these are dynamically linked with the running kernel through an *eager* binding approach that does not involve `.got`, `.plt`, and similar constructs [78]; (c) have limited support for JIT code in kernel space (typically to facilitate tracing and packet filtering [40]). These reasons prompted us to study new leakage-resilient diversification schemes, fine-tuned for the kernel.

3.4 Design

3.4.1 R^X Enforcement

kR^X employs a self-protection approach to R^X, inspired by software fault isolation (SFI) [116, 140, 172, 190, 199]. However, there is a fundamental difference between previous work on SFI and kR^X: SFI tries to *sandbox untrusted code*, while kR^X *read-protects benign code.* SFI schemes (e.g., PittSFIeld [140], NaCl [172, 199]) are designed for confining the control flow and memory write operations of the sandboxed code, typically by imposing a canonical layout [172], bit-masking memory writes [190], and instrumenting computed branch instructions [140]. The end goal of SFI is to limit memory corruption in a subset of the address space, and ensure that execution does not escape the sandbox [199].

In contrast, kR^X focuses on the *read* operations of benign code that can be abused to disclose memory [121]. (Memory reads are usually ignored by conventional SFI schemes, due to the non-trivial overhead associated with their instrumentation [17, 140].) However, the difference between our threat model and that of SFI allows us to make informed design choices and implement a set of optimizations that result in R^X enforcement with low overhead. We explore the full spectrum of settings and trade-offs, by presenting: (a) `kR^X-SFI`: a software-only R^X scheme; (b) `kR^X-MPX`: a hardware-assisted R^X scheme, which exploits the Intel Memory Protection Extensions (MPX) [102] to (almost) eliminate the protection overhead; (c) `kR^X-SEG`: a hardware-based R^X scheme that leverages memory segmenta-

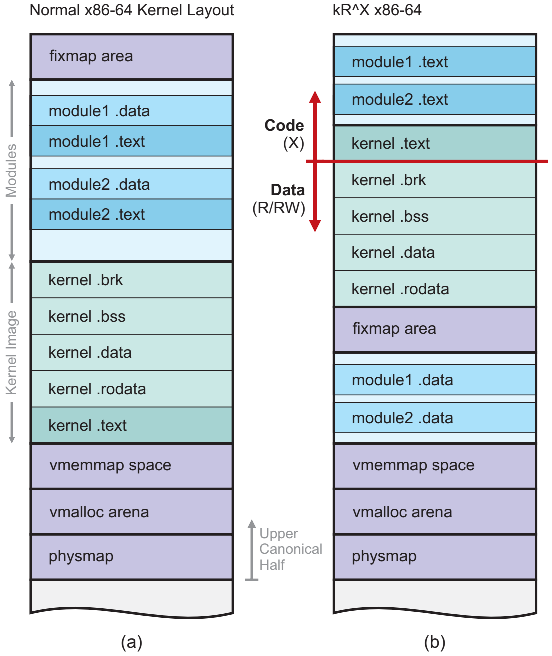


Figure 3.1: The Linux kernel space layout in x86-64: (a) vanilla and (b) `kR^X-KAS`. The kernel image and `modules` regions may contain additional (ELF) sections; only the standard ones are shown.

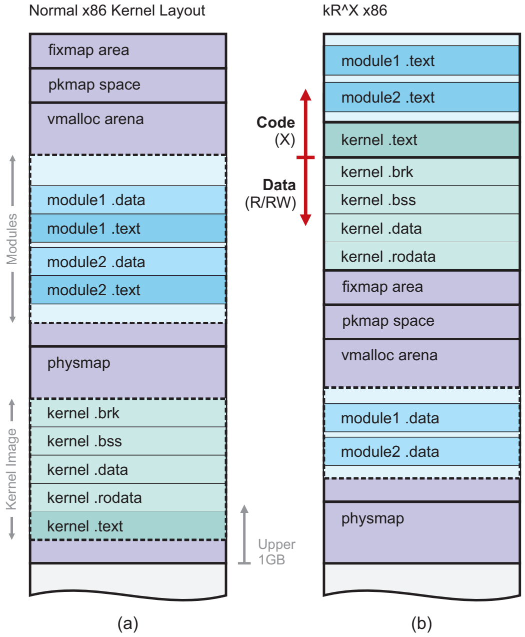


Figure 3.2: The Linux kernel space layout in x86 (under the default 3G/1G user/kernel split): (a) vanilla and (b) `kR^X-KAS`. The kernel image and `modules` regions may contain extra sections.

tion (available in legacy systems) [101]; and (d) `kR^X-KAS`: a new kernel space layout that facilitates the efficient R^X enforcement by (a), (b), and (c).

3.4.1.1 `kR^X-KAS` (x86 & x86-64)

The x86-64 architecture uses 48-bit virtual addresses that are sign-extended to 64 bits (bits `[48:63]` are copies of bit `[47]`), splitting the 64-bit virtual address space in two halves of 128TB each. In x86-64 Linux, kernel space occupies the upper canonical half (`[0xFFFF800000000000:`$2^{64}-1$`]`), and is further divided into six regions (see Figure 3.1(a)) [113]: `fixmap`, `modules`, kernel image, `vmemmap` space, `vmalloc` arena, and `physmap`. In x86 Linux, kernel space can be assigned to the upper 1GB, 2GB, or 3GB part of the virtual address space, with the first option being the default (3G/1G split). However, as address space is limited in 32-bit platforms, different regions collide to prevent waste (e.g., kernel image and physmap, modules and vmalloc arena; see Figure 3.2(a)) [108].

Unfortunately, the default layout does not promote the enforcement of R^X, as it blends together code and data regions. To facilitate a *unified* and efficient treatment by our different enforcement mechanisms (SFI, MPX, SEG), kR^X relies on a modified kernel layout that maps code and data into *disjoint*, contiguous regions (see Figure 3.1(b); x86-64, and Figure 3.2(b); x86). The code region is carved from the top part of kernel space with its exact size being controlled by the __START_KERNEL_map configuration option. All other regions are left unchanged, except fixmap (and pkmap in x86), which is "pushed" towards lower addresses, and modules, which is replaced by two newly-created areas: modules_text and modules_data. modules_text occupies the original modules area, whereas modules_data is placed right below fixmap. The size of both regions is configurable, with the default value set to 512MB in x86-64, and 256MB in x86.[2]

3.4.1.2 Kernel Image

The kernel image is loaded in its assigned location by a staged bootstrap process. Conventionally, the .text section is placed at the beginning of the image, followed by standard (i.e., .rodata, .data, .bss, .brk) and kernel-specific sections [15]. kR^X revamps (flips) this layout by placing .text at the end of the ELF object. Hence, during boot time, after vmlinuz is copied in memory and decompressed, .text lands at the code region of kR^X-KAS; all other sections end up in the data region.[3] The symbols _krx_edata and _text denote the end of the data region and the beginning of the code region, in kR^X-KAS.

3.4.1.3 Kernel Modules

Although kernel modules (.ko files) are also ELF objects, their on-disk layout is left unaltered by kR^X, as the separation of .text from all other (data) sections occurs during load time. A kR^X-KAS-aware module loader-linker slices the .text section and copies it in

[2] The default setting was selected by dividing the original modules area in two equally-sized parts.

[3] Note that __ex_table, __tracepoints, __jump_table, and every other similar section that contains mostly (in)direct code pointers, are placed at the code (non-readable) region and marked as non-executable.

`modules_text`, while the rest of the (allocatable) sections of the ELF object are loaded in `modules_data`. Once everything is copied in kernel space, relocation and symbol binding take place (eager loading [14]).

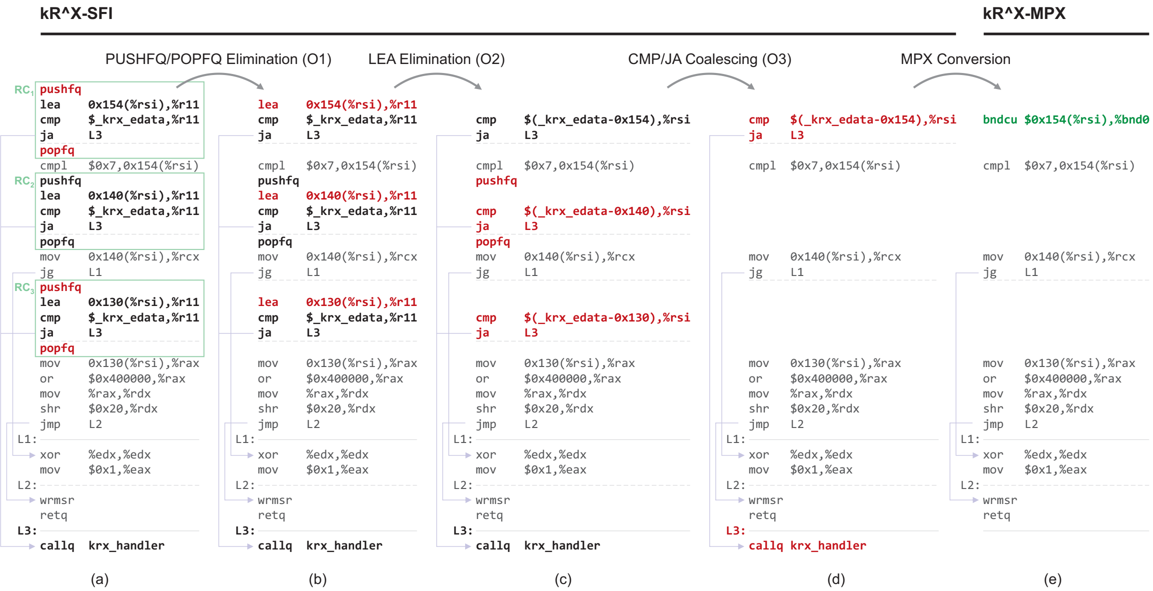
3.4.1.4 Physmap

The `physmap` area is a contiguous kernel region that contains a *direct* (1:1) mapping of all physical memory to facilitate dynamic kernel memory allocation [108]. Hence, as physical memory is alloted to the kernel image and modules, the existence of `physmap` results in *address aliasing*; virtual address aliases, or *synonyms* [115], occur when two (or more) different virtual addresses map to the same physical memory address. Consequently, kernel code becomes accessible not only through the code region (virtual addresses above `_text`), but also via `physmap`-resident code synonyms in the data region. To deal with this issue, kR^X always unmaps any synonym pages of `.text` sections from `physmap` (as well as synonym pages of any other section that resides in the code region), and maps them back whenever modules are unloaded (after zapping their contents to prevent code layout inference attacks [175]).

3.4.1.5 Alternative Layouts

`kR^X-KAS` has several advantages over the address space layouts imposed by SFI-based schemes (e.g., NaCl [199], LR^2 [17]). First, address space waste is kept to a minimum; LR^2 chops the address space in half to enforce a policy similar to R^X, whereas `kR^X-KAS` mainly *rearranges* sections. More importantly, in 32-bit systems, a smaller kernel space would necessitate the use of `kmap`/`kunmap` operations for managing page frames that cannot be directly addressed through `physmap` [108],[4] which, in turn, translates to higher runtime overhead; `kmap`/`kunmap` operations require altering the kernel page table, resulting in TLB pressure [145] and shootdowns. Second, the use of bit-masking confinement (similarly to NaCl [199] and LR^2 [17]), in the kernel setting, requires a radically different set of memory allocators to cope with the alignment constrains of bit-masking. In contrast, the layout of

[4] To access the contents of a page frame, the kernel must first map that frame in kernel space. In x86, the kernel has only 1GB – 3GB virtual addresses available for managing (up to) 64GB of RAM.



(a)

```
    pushfq
    lea     0x154(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3
    popfq
    cmpl    $0x7,0x154(%rsi)
    pushfq
    lea     0x140(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3
    popfq
    mov     0x140(%rsi),%rcx
    jg      L1
    pushfq
    lea     0x130(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3
    popfq
    mov     0x130(%rsi),%rax
    or      $0x400000,%rax
    mov     %rax,%rdx
    shr     $0x20,%rdx
    jmp     L2
L1:
    xor     %edx,%edx
    mov     $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq   krx_handler
```

(b)

```
    lea     0x154(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3

    cmpl    $0x7,0x154(%rsi)
    pushfq
    lea     0x140(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3
    popfq
    mov     0x140(%rsi),%rcx
    jg      L1

    lea     0x130(%rsi),%r11
    cmp     $_krx_edata,%r11
    ja      L3

    mov     0x130(%rsi),%rax
    or      $0x400000,%rax
    mov     %rax,%rdx
    shr     $0x20,%rdx
    jmp     L2
L1:
    xor     %edx,%edx
    mov     $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq   krx_handler
```

(c)

```
    cmp     $(_krx_edata-0x154),%rsi
    ja      L3

    cmpl    $0x7,0x154(%rsi)
    pushfq

    cmp     $(_krx_edata-0x140),%rsi
    ja      L3
    popfq
    mov     0x140(%rsi),%rcx
    jg      L1

    cmp     $(_krx_edata-0x130),%rsi
    ja      L3

    mov     0x130(%rsi),%rax
    or      $0x400000,%rax
    mov     %rax,%rdx
    shr     $0x20,%rdx
    jmp     L2
L1:
    xor     %edx,%edx
    mov     $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq   krx_handler
```

(d)

```
    cmp     $(_krx_edata-0x154),%rsi
    ja      L3

    cmpl    $0x7,0x154(%rsi)

    mov     0x140(%rsi),%rcx
    jg      L1

    mov     0x130(%rsi),%rax
    or      $0x400000,%rax
    mov     %rax,%rdx
    shr     $0x20,%rdx
    jmp     L2
L1:
    xor     %edx,%edx
    mov     $0x1,%eax
L2:
    wrmsr
    retq
L3:
    callq   krx_handler
```

(e)

```
    bndcu   $0x154(%rsi),%bnd0

    cmpl    $0x7,0x154(%rsi)

    mov     0x140(%rsi),%rcx
    jg      L1

    mov     0x130(%rsi),%rax
    or      $0x400000,%rax
    mov     %rax,%rdx
    shr     $0x20,%rdx
    jmp     L2
L1:
    xor     %edx,%edx
    mov     $0x1,%eax
L2:
    wrmsr
    retq
```

Figure 3.3: The different optimization phases of `kR^X-SFI` (a)–(d) and `kR^X-MPX` (e).

`kR^X-KAS` is transparent to the kernel's performance-critical allocators [13]. Third, important kernel features that are tightly coupled with the kernel address space, like KASLR [59] or alternative user/kernel splits (e.g., 2G/2G, 1G/3G) [35], are readily supported without requiring any kernel code change or redesign.

Finally, in x86-64, the *code model* (`-mcmodel=kernel`) used generates code for the negative 2GB of the address space [77]. This model requires the `.text` section of the kernel image and modules, and their respective global data sections, to be not more than 2GB apart. The reason is that the offset of the x86-64 `%rip`-relative `mov` instructions is only 32 bits. `kR^X-KAS` respects this constraint, whereas a scheme like LR^2 (halved address space) would require transitioning to `-mcmodel=large`, which incurs additional overhead, as it rules out `%rip`-relative addressing. Interestingly, the development of `kR^X-KAS` helped uncover two kernel bugs (Appendix A).

3.4.2 `kR^X-SFI` (x86-64)

`kR^X-SFI` is a software-only R^X scheme that targets modern (64-bit) platforms. Once the `kR^X-KAS` layout is in place, R^X can be enforced by checking *all* memory reads and making sure they fall within the data region (addresses below `_krx_edata`). As bit-masking load instructions is not an option, due to the non-canonical layout, `kR^X-SFI` employs *range checks* (RCs) instead. The range checks are placed (at compile time) right before memory read operations, ensuring (at runtime) that the *effective addresses* of reads are valid. We will be using the example code of Figure 3.3 to present the internals of `kR^X-SFI`. The original code excerpt is listed in Figure 3.3(e) (excluding the `bndcu` instruction at the function prologue) and is from the `nhm_uncore_msr_enable_event()` routine of the x86-64 Linux kernel (v3.19, GCC v4.7.2) [133]. It involves three memory reads: `cmpl $0x7,0x154(%rsi)`; `mov 0x140(%rsi),%rcx`; and `mov 0x130(%rsi),%rax`.

We begin with a basic, unoptimized (`O0`) range check scheme, and continue with a series of optimizations (`O1`–`O3`) that progressively rectify the RCs for performance. Note that similar techniques are employed by SFI systems [140, 172, 190], but earlier work focuses on RISC-based architectures [17, 190] or fine tunes bit-masking confinement [140]. We study the problem in a CISC (x86-64) setting, and introduce a principled approach to optimize checks on memory reads operating on non-canonical layouts.

3.4.2.1 Basic Scheme (`O0`)

`kR^X-SFI` prepends memory read operations with a range check implemented as a sequence of five instructions, as shown in Figure 3.3(a). First, the effective address of the memory read is loaded by `lea` in the `%r11` scratch register, and is subsequently checked against the end of the data region (`cmp`). If the effective address falls above `_krx_edata` (`ja`), then this is a R^X violation, as the read tries to access the code region. In this case, `krx_handler()` is invoked (`callq`) to handle the violation; our default handler appends a warning message to the kernel log and halts the system, but stringent policies, like *active kernel exploit response* [90], can also be supported. Finally, to preserve the semantics of the original control flow, the [`lea`, `cmp`, `ja`] triplet is wrapped with `pushfq` and `popfq` to maintain the value of `%rflags`, which is altered by `cmp`.

3.4.2.2 pushfq/popfq Elimination (`O1`)

Spilling and filling the `%rflags` register is expensive [137]. However, we can eliminate *redundant* `pushfq`-`popfq` pairs by performing a liveness analysis on `%rflags`. Figure 3.3(b) depicts this optimization. Every `cmp` instruction of a range check starts a new live region for `%rflags`. If there are no kernel instructions that use `%rflags` inside a region, we can avoid preserving it. For example, in Figure 3.3(b), RC_1 is followed by a `cmpl` instruction that starts a new live region for `%rflags`. Hence, the live region defined by the `cmp` instruction of RC_1 contains no original kernel instructions, allowing us to safely eliminate `pushfq-popfq` from RC_1. Similarly, the live region started by the `cmp` instruction of RC_3 reaches only `mov 0x130(%rsi),%rax`, as the subsequent `or` instruction redefines `%rflags` and starts a new live region. As `mov` does not use `%rflags`, `pushfq-popfq` can be removed from RC_3. The `cmp` instruction of RC_2, however, starts a live region for `%rflags` that reaches `jg L1`—a jump instruction that depends on `%rflags`—and thus `pushfq-popfq` are not eliminated from RC_2. This optimization can eliminate up to 94% of the original `pushfq-popfq` pairs (see Section 3.6.2).[5]

3.4.2.3 lea Elimination (`O2`)

If the effective address of a read operation is computed using only a base register and a displacement, we can further optimize our range checks by eliminating the `lea` instruction and *adjusting* the operands of the `cmp` instruction accordingly. That is, we replace the scratch register (`%r11`) with the base register (`%reg`), and modify the end of the data region by adjusting the displacement (`offset`). Note that both RC schemes are computationally equivalent. Figure 3.3(c) illustrates this optimization. In all cases `lea` instructions are eliminated, and `cmp` is adjusted accordingly. Marked, 95% of the RCs can be optimized this way.

3.4.2.4 cmp/ja Coalescing (`O3`)

Given two RCs, RC_a and RC_b, which confine memory reads that use the same base register (`%reg`) and different displacements ($\texttt{offset}_a$ `!=` $\texttt{offset}_b$), we can *coalesce* them to one RC that checks against the maximum displacement, if in all control paths between RC_a and

[5] We do not track the use of individual bits (status flags) of `%rflags`. As long as a kernel instruction, inside a live region, uses *any* of the status bits, we preserve the value of `%rflags`—even if that instruction uses a bit not related to the one(s) modified by the RC `cmp` (i.e., we over-preserve).

RC_b %reg is never: (a) redefined; (b) spilled to memory. Note that by recursively applying the above in a routine, until no more RCs can be coalesced, we end up with the *minimum* set of checks required to confine every memory read.

Figure 3.3(d) illustrates this optimization. All memory operations protected by the checks RC_1, RC_2, and RC_3 use the same base register (%rsi), but different displacements (0x154, 0x140, 0x130). As %rsi is never spilled, filled, or redefined in any path between RC_1 and RC_2, RC_1 and RC_3, and RC_2 and RC_3, we coalesce all range checks to a single RC that uses the maximum displacement, confining all three memory reads. If %rsi + 0x154 < _krx_edata, then %rsi + 0x140 and %rsi + 0x130 are guaranteed to "point" below _krx_edata, as long as %rsi does not change between the RC and the respective memory reads. The reason we require %rsi not to be spilled is to prevent temporal attacks, like those demonstrated by Conti et al. [31]. About one out of every two RCs can be eliminated using RC coalescing.

3.4.2.5 Stack Reads

If the stack pointer (%rsp) is used with a scaled index register [101], the read is instrumented with a range check as usual. However, if the effective address of a stack read consists only of (%rsp) or offset(%rsp), the range check can be eliminated by spacing appropriately the code and data regions. Recall, though, that attackers may *pivot* %rsp anywhere inside the data region. By repeatedly positioning %rsp at (or close to) _krx_edata, they could take advantage of uninstrumented stack reads and leak up to offset bytes from the code region (assuming they control the contents at, or close to, _krx_edata for reconciling the effects of the dislocated stack pointer). kR^X-SFI deals with this slim possibility by placing a guard section (namely .krx_phantom), between _krx_edata and the beginning of the code region. Its size is set to be greater than the maximum offset of all %rsp-based memory reads.

3.4.2.6 String Operations and Safe Reads

The x86 string operations [101], namely cmps, lods, movs, and scas, read memory via the %rsi register (except scas, which uses %rdi). kR^X-SFI instruments these instructions with RCs that check (%rsi) or (%rdi), accordingly. If the string operation is rep-prefixed, the RC is placed *after* the confined instruction, checking `%rsi` (or `%rdi`) once the respective operation is complete.[6] Lastly, absolute and `%rip`-relative memory reads are not instrumented with range checks, as their effective addresses are encoded within the instruction itself and cannot be modified at runtime due to W^X. Safe reads account for 4% of all memory reads.

3.4.3 `kR^X-MPX` (x86-64)

`kR^X-MPX` is a hardware-assisted, R^X scheme that takes advantage of the MPX (Memory Protection Extensions) feature [102], available in the latest Intel CPUs, to enforce the range checks and nearly eliminate their runtime overhead. To the best of our knowledge, kR^X is the first system to exploit MPX for confining memory reads and implementing a memory safety policy (R^X) within the OS.[7]

MPX introduces four new bounds registers (`%bnd0`–`%bnd3`), each consisting of two 64-bit parts (`lb`; lower bound, `ub`; upper bound). `kR^X-MPX` uses `%bnd0` to implement RCs and initializes it as follows: `lb = 0x0` and `ub = _krx_edata`, effectively covering everything up to the end of the data region. Memory reads are prefixed with a RC as before (at compile time), but the [`lea`, `cmp`, `ja`] triplet is now replaced with a *single* MPX instruction (`bndcu`), which checks the effective address of the read against the upper bound of `%bnd0`. Figure 3.3(e) illustrates the instrumentation performed by `kR^X-MPX`. Note that `bndcu` does not alter `%rflags`, so there is no need to preserve it. Also, the checked effective address is encoded in the MPX instruction itself, rendering the use of `lea` with a scratch register unnecessary, while violations trigger a CPU exception (`#BR`), obviating the need to invoke `krx_handler()` explicitly. In a nutshell, optimizations `O1` and `O2` are not relevant when MPX is used to implement range checks, whereas `O3` (RC coalescing) is used as before.

[6] We generate `rep`-prefix string instructions that operate on ascending memory addresses (`%rflags.df = 0`). By placing the RC immediately after the confined instruction, we can still identify reads from the code region, albeit postmortem, without breaking code optimizations.

[7] Interestingly, although the Linux kernel already includes the necessary infrastructure to provide MPX support in user programs, kernel developers are reluctant to use MPX for the kernel itself [41].

Lastly, the user mode value of %bnd0 is spilled and filled on every mode switch; kR^X-MPX does not interfere with the use of MPX by user applications.

3.4.4 kR^X-SEG (x86)

In legacy (32-bit) systems, kR^X-SEG enforces the R^X policy using memory segmentation [101]. Note that the use of segmentation for isolation purposes has been well researched, both in user space [199] and kernel space [154] settings. Nevertheless, we do present the design of a segmentation-based R^X scheme for completeness, and for demonstrating that kR^X's memory layout enables a *unified* R^X treatment by both software-based (SFI, MPX) and hardware-only (SEG) schemes.

As x86 forbids disabling segmentation completely, Linux uses flat code and data segments that cover the whole 32-bit address space (4GB), neutralizing its effect. kR^X-SEG redefines the kernel data segment(s) to be in par with the data region of kR^X-KAS. That is, the base address of the segment remains 0x0, whereas its limit is set to _krx_edata >> PAGE_SHIFT[8], effectively turning every access to the code region (i.e., addresses above _krx_edata) into a protection fault (#GP). kR^X-SEG redefines the DS, ES, and FS (per-CPU data) segments; CS is left flat as it is not involved in data accesses, GS is only used by the stack-smashing protector [159, 186], and limited to 4 bytes (by default), whereas SS is left flat as well because of .krx_phantom (see "Stack Reads" in Section 3.4.2). Note that in contrast to kR^X-{SFI, MPX}, kR^X-SEG enforces the R^X policy without relying on (kernel) code instrumentation.

3.4.5 Fine-grained KASLR

With kR^X-{SFI, MPX, SEG} ensuring the secrecy of kernel code under the presence of arbitrary memory disclosure, the next step for the prevention of (JIT-)ROP/JOP is the diversification of the kernel code itself—if not coupled with code diversification, any execute-only defense is useless [31, 54]. The use of code perturbation or randomization to hinder code-reuse attacks has been studied extensively in the past [9, 57, 85, 96, 111, 119, 150, 193]. Previous research, however, either did not consider resilience to indirect JIT-ROP [31, 54],

[8] PAGE_SHIFT = lg(PAGE_SIZE) (i.e., 12 for 4KB pages).

or focused on schemes geared towards userland code [17, 48]. kR^X introduces code diversification designed from the ground up to mitigate both *direct* and *indirect* (JIT-)ROP/JOP attacks for the kernel setting.

3.4.5.1 Foundational Diversification

kR^X diversifies code through a recursive process that permutes chunks of code. The end goal of our approach is to fabricate kernel (`vmlinux`) images and `.ko` files (modules) with no gadgets left at predetermined locations. At the function level, we employ code block randomization [57, 193], whereas at the section (`.text`) level, we perform function permutation [9, 111].

3.4.5.2 Phantom Blocks

Slicing a function into arbitrary code blocks and randomly permuting them results (approximately) in $lg(B!)$ bits of entropy, where B is the number of code blocks [57]. However, as the achieved randomness depends on B, routines with a few basic blocks end up having extremely low randomization entropy. For instance, ∼12% of the Linux kernel's (v3.19, GCC v4.7.2) routines consist of a single basic block (i.e., zero entropy). We note that this issue has been overlooked by previous studies [57, 193], and we augmented kR^X to resolve it as follows.

Starting with `k`, the number of randomization entropy bits *per function* we seek to achieve (a compile-time parameter), we first slice routines at call sites (i.e., code blocks ending with a `call` instruction). If the resulting number of code blocks does not allow for `k` (or more) bits of entropy, we further slice each code block according to its basic blocks. If the achieved entropy is still not sufficient, we pad routines with fake code blocks, dubbed *phantom blocks*, filled with a random number of `int 3` instructions (stepping on them triggers a CPU exception; `#BR`). Having achieved adequate slicing, kR^X randomly permutes the final code and phantom blocks and "patches" the Control Flow Graph (CFG), so that the original control flow remains unaltered. Any phantom blocks, despite being mixed with regular code, are never executed due to properly-placed `jmp` instructions. Our approach attains the desired randomness with the *minimum* number of code cuts and padding.

3.4.5.3 Function Entry Points

Without code block permutation, an attacker that discloses a function pointer can still reuse gadgets from the entry code block of the respective function. To prevent this, functions always begin with a phantom block: the first instruction of each function is a `jmp` instruction that transfers control to the original first code block. Hence, an attacker armed with a leaked function pointer can only reuse a whole function, which is not a viable strategy, as function arguments in both x86 and x86-64 Linux kernels are passed through registers [20, 142]. Consequently, as we further discuss in Section 3.6.3.3, attackers must first use gadgets to initialize the appropriate registers before invoking a function.

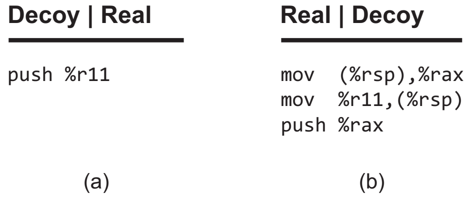
3.4.5.4 Return Address Protection

Return addresses are stored in kernel stacks, which are allocated from the readable data (`physmap`) region of `kR^X-KAS` [108]. Conti et al. demonstrated an indirect JIT-ROP attack that relies on harvesting return addresses from stacks [31]. kR^X treats return addresses specially to mitigate such indirect JIT-ROP attempts.

3.4.5.5 Return Address Encryption (`X`)

We employ an `XOR`-based encryption scheme to protect saved return addresses from being disclosed [17, 157, 195]. Every routine is associated with a secret key (`xkey`), placed in the non-readable region of `kR^X-KAS`, while function prologues and epilogues are instrumented as follows: `mov offset(%rip),%r11; xor %r11,(%rsp)`. That is, `xkey` is loaded into a scratch register (`%r11`), which is subsequently used to encrypt or decrypt the saved return address. The `mov` instruction that loads `xkey` from the code region is `%rip`-relative (safe read), and hence not affected by kR^X. In x86, where `%rip`-relative addressing is not available, `mov` instructions are prefixed with the `%ss` selector (recall that `kR^X-SEG` retains a flat 4GB `SS` segment), and their (memory read) operand is replaced with the absolute address corresponding to `xkey`; the scratch register used in x86 is `%esi`.

In summary, unmangled return addresses are pushed into the kernel stack by the caller (`call`), encrypted by the callee, and remain encrypted until the callee returns (`ret`) or performs a tail call. In the latter case, the return address is temporarily decrypted by the



```
push %r11
```

```
mov  (%rsp),%rax
mov  %r11,(%rsp)
push %rax
```

Figure 3.4: Instrumentation code (function prologue; x86-64) to place the decoy return address (a) below or (b) above the real one.

function that is about to tail-jump, and re-encrypted by the new callee. Return sites are also instrumented to zap decrypted return addresses. Note that the `xkey` variables are initialized with a random value at compile time, and merged into a contiguous region at link time. At boot time, once the kernel initializes its entropy pool(s), the respective `xkey` variables of the kernel image are replenished with new random values, whereas upon loading kernel modules, the module loader-linker places the corresponding `xkey` variables in the protected region and also replenishes them with random values.

3.4.5.6 Return Address Decoys (D)

Return address decoys are an alternative scheme that leverages *deception* to mitigate the disclosure of return addresses. The main benefit over return address encryption is their slightly lower overhead in some settings, as discussed in Section 3.6.2. We begin with the concept of *phantom instructions*, which is key to return address decoys. Phantom instructions are effectively `NOP` instructions that contain overlapping "tripwire" (e.g., `int 3`) instructions, whose execution raises an exception [47].

For instance, `mov $0xcc,%r11` (`mov $0xcc,%esi` in x86) is a phantom instruction; apart from changing the value of `%r11` (`%esi`), it does not alter the CPU or memory state. The opcodes of the instruction are the following: `49 C7 C3 CC 00 00 00` in x86-64, or `BE CC 00 00 00` in x86. Note that `0xCC` is also the opcode for `int 3`, which raises a `#BR` exception when executed. kR^X pairs every return site in a routine with the tripwire of a separate phantom instruction, randomly placed in the respective routine's code stream. Call sites are instrumented to pass the address of the tripwire to the callee through a predetermined scratch register (i.e., %r11 in x86-64, %esi in x86). Armed with that information, the callee either: (a) places the address of the tripwire right below the saved return address on the stack; or (b) relocates the return address so that the address of the tripwire is stored where the return address used to be, followed by the saved return address (Figure 3.4 illustrates the concept in x86-64). In both cases, the callee stores two addresses sequentially on the stack. One is the real return address (R) and the other is the decoy one (D).[9] The exact ordering is decided randomly at compile time.

kR^X always slices routines at call sites. Therefore, by randomly inserting phantom instructions in routine code, their relative placement to return sites cannot be determined in advance (code block randomization perturbs them independently). As a result, although return address-decoy pairs can be harvested from the kernel stack(s), the attacker cannot differentiate which is which, because that information is encoded in each routine's code, which is not readable (R^X). The net result is that call-preceded gadgets [23, 55, 86] are coupled with a *pair* of return addresses (R and D), thereby forcing the attacker to randomly choose one of them. If n call-preceded gadgets are required for an indirect JIT-ROP attack, the attacker will succeed (i.e., correctly guess the real return address in all cases) with a probability $\mathtt{P_{succ}} = \mathtt{1/2^n}$.

3.5 Implementation

3.5.1 Toolchain

We implemented kR^X-{SFI, MPX, SEG} as a set of modifications to the pipeline of GCC v4.7.2—the "de facto" C compiler for building Linux. Specifically, we instrumented the intermediate representation (IR) used during translation to: (a) perform the RC-based (R^X) confinement (see Section 3.4.2 and 3.4.3); and (b) randomize code blocks and protect return addresses (Sections 3.4.5.1 and 3.4.5.4). Our prototype consists of two plugins, krx

[9] Stack offsets are adjusted whenever necessary: if frame pointers are used, negative %{r,e}bp offsets are decreased by sizeof(unsigned long); if frame pointers are omitted, %{r,e}sp-based accesses to non-local variables are increased by sizeof(unsigned long). Function epilogues, depending on the scheme employed, make use of the real return address (i.e., by adjusting %{r,e}sp before ret and tail calls).

and `kaslr`. The `krx` plugin is made up of 5 KLOC and `kaslr` of 12 KLOC (both written in C), resulting in two position-independent (PIC) dynamic shared objects, which can be loaded to GCC with the `-fplugin` directive.

We chain the instrumentation of `krx` after the `vartrack RTL` optimization pass, by calling GCC's `register_callback()` function and hooking with the pass manager [109]. The reasons for choosing to implement our instrumentation logic at the `RTL` level, and not as annotations to the `GENERIC` or `GIMPLE` IR, are the following. First, by applying our instrumentation after the important optimizations have been performed, which may result into instructions being moved or transformed, it is guaranteed that only relevant code will be protected. Second, any implicit memory reads that are exposed later in the translation process are not neglected. Third, the inserted range checks are tightly coupled with the corresponding unsafe memory reads. This way, the checks are protected from being removed or shifted away from the respective read operations, due to subsequent optimization passes [31].

The `kaslr` plugin is chained *after* `krx`, or after `vartrack` if `krx` is not loaded. Code block slicing and permutation is the final step, after the R^X instrumentation and return address protection. By default, `krx` implements the `kR^X-SFI` scheme, operating at the maximum optimization level (`O3`); `kR^X-MPX` can be enabled with the following knob: `-fplugin-arg-krx-mpx=1`. Likewise, `kaslr` uses the `XOR`-based encryption scheme by default, and sets `k` (the number of entropy bits per-routine; see Section 3.4.5.4) to 30. Return address decoys can be enabled with `-fplugin-arg-kaslr-dec=1`, while `k` may be adjusted using `-fplugin-arg-kaslr-k=N`.

3.5.2 Kernel Support

`kR^X-KAS` (Section 3.4.1.1) and `kR^X-SEG` (Section 3.4.4) are implemented as a set of patches (∼10 KLOC) for the Linux kernel (v3.19), which perform the following changes: (a) construct `kR^X-KAS` by adjusting the kernel page tables (`init_level4_pgt`, `swapper_pg_dir`); (b) make the module loader-linker `kR^X-KAS`-aware; (c) (un)map certain synonyms from `physmap` during kernel bootstrap and module (un)loading; (d) replenish `xkey` variables during initialization (only if `XOR`-based encryption is used); (e) set the limit of `DS`, `ES`, and FS segments to _krx_edata >> PAGE_SHIFT in gdt_page (x86 SEG only); (f) reserve %bnd0, load it with the value of _krx_edata, and spill/fill it on mode switches (MPX only); (g) place .text section(s) at the end of the vmlinux image and permute their functions (vmlinux.lds.S); (h) map the kernel image in kR^X-KAS, so that executable code resides in the non-readable region. Note that although kR^X requires patching the OS kernel, and (re)compiling with custom GCC plugins, it does support *mixed* code: i.e., both protected and unprotected modules; this design not only allows for incremental deployment and adoption, but also facilitates selective hardening [81].

3.5.3 Assembly Code

Both krx and kaslr are implemented as RTL IR optimization passes, and, therefore, cannot handle *assembly* code (both "inline" or external). However, this is not a fundamental limitation of kR^X, but rather an implementation decision. In principle, the techniques presented in Section 3.4.1 and 3.4.5 can all be incorporated in the assembler, instead of the compiler, as they do not depend on high-level semantics.

3.5.4 Legitimate Code Reads

Kernel tracing and debugging (sub)systems, like ftrace and KProbes [40], as well as the module loader-linker, need access to the kernel code region. To provide support for such frameworks, we cloned seven functions of the get_next and peek_next family of routines, as well as memcpy, memcmp, and bitmap_copy; the cloned versions of these ten functions are not instrumented by the krx GCC plugin—they are instrumented, however, by the kaslr GCC plugin, and thus their callers' return addresses are protected and their code is randomized accordingly. Lastly, ftrace, KProbes, and the module loader-linker, were patched to use the kR^X-based versions (i.e., the clones) of these functions (∼330 LOC), and care was taken to ensure that none of them is leaked through function pointers or the symbol table of the kernel.

3.5.5 Forward Porting

Porting kR^X to newer (v4.x) kernel versions requires moderate engineering effort. More specifically, two recent kernel features that demand special handling are: (a) BPF JIT [38] and (b) live kernel patching [42]. To provide support for the former, the BPF JIT compiler needs to be extended to include the techniques presented in Section 3.4.1 and 3.4.5, and also place the emitted code in the non-readable region of `kR^X-KAS`. To provide support for the latter, any routine that belongs to the patching framework, and requires reading kernel code, needs to be treated similarly to `ftrace`, KProbes, *etc.* (see “Legitimate Code Reads” above).

3.6 Evaluation

We studied the runtime overhead of `kR^X-{SFI, MPX, SEG}`, both as standalone implementations, as well as when applied in conjunction with the code randomization schemes described in Section 3.4.5 (i.e., fine-grained KASLR coupled with return address encryption or return address decoys). We used the LMBench suite [141] for micro-benchmarking, and employed the Phoronix Test Suite (PTS) [162] to measure the performance impact on real-world applications. (Note that PTS is used by the Linux kernel developers to track performance regressions.) The reported results are average values of ten and (at least) five runs, respectively, and all benchmarks were used with their default settings. To obtain a representative sample when measuring the effect of randomization schemes, we compiled the kernel ten times, using an identical configuration, and averaged the results.

3.6.1 Testbed

Our experiments were carried out on a Debian GNU/Linux v7 system, equipped with a 4GHz quad-core Intel Core i7-6700K (Skylake) CPU and 16GB of RAM. The kR^X plugins were developed for GCC v4.7.2, which was also used to build all Linux kernels (v3.19) with the default configuration of Debian (i.e., including all modules and device drivers). Lastly, the kR^X-protected kernels were linked and assembled using `binutils` v2.25.

3.6.2 Performance

3.6.2.1 Micro-benchmarks

To assess the impact of kR^X on the various kernel subsystems and services we used LMBench [141], focusing on two metrics: *latency* and *bandwidth* overhead. Specifically, we measured the additional latency imposed on: (a) critical system calls, like `open()`/`close()`, `read()`/`write()`, `select()`, `fstat()`, `mmap()`/`munmap()`; (b) mode switches (i.e., user mode to kernel mode and back) using the `null` system call; (c) process creation (`fork()+exit()`, `fork()+execve()`, `fork()+/bin/sh`); (d) signal installation (via `sigaction()`) and delivery; (e) protection faults and page faults; (f) pipe I/O and socket I/O (`AF_UNIX` and `AF_INET` TCP/UDP sockets). Moreover, we measured the bandwidth degradation on pipe, socket (`AF_UNIX` and `AF_INET` TCP), and file I/O.

Table 3.1 summarizes our results on x86-64. The columns `SFI(-O0)`, `SFI(-O1)`, `SFI(-O2)`, `SFI(-O3)`, and `MPX` correspond to the overhead of RC-based (R^X) confinement. In addition, `SFI(-O0)`–`SFI(-O3)` illustrate the effect of `pushfq/popfq` elimination, `lea` elimination, and `cmp/ja` coalescing, when applied on an aggregate manner. The columns `D` and `X` correspond to the overhead of return address protection (`D`: return address decoys, `X`: return address encryption) *coupled* with fine-grained KASLR, whereas the last four columns (`SFI+D`, `SFI+X`, `MPX+D`, `MPX+X`) report the overhead of the full protection schemes that kR^X provides.

The software-only `kR^X-SFI` scheme incurs an overhead of up to 24.82% (avg. 10.86%) on latency and 6.43% (avg. 2.78%) on bandwidth. However, with hardware support (`kR^X-MPX`) the respective overheads decrease dramatically; latency: ≤ 6.27% (avg. 1.35%), bandwidth: ≤ 1.43% (avg. 0.34%). The overhead of fine-grained KASLR is relatively higher; when coupled with return address decoys (`D`), it incurs an overhead of up to 15.03% (avg. 6.21%) on latency and 3.71% (avg. 1.66%) on bandwidth; when coupled with return address encryption (`X`), it incurs an overhead of up to 18.3% (avg. 9.3%) on latency and 4.4% (avg. 3.71%) on bandwidth. Lastly, the overheads of the full kR^X protection schemes translate (roughly) to the sum of the specific R^X enforcement mechanism (`kR^X-SFI`, `kR^X-MPX`) and fine-grained KASLR scheme (`D`, `X`) used.

	Benchmark	SFI(-O0)	SFI(-O1)	SFI(-O2)	SFI(-O3)	MPX	D	X	SFI+D	SFI+X	MPX+D	MPX+X
Latency	syscall()	126.90%	13.41%	13.44%	12.74%	0.49%	0.62%	2.70%	13.67%	15.91%	2.24%	2.92%
	open()/close()	306.24%	39.01%	37.45%	24.82%	3.47%	15.03%	18.30%	40.68%	44.56%	19.44%	22.79%
	read()/write()	215.04%	22.05%	19.51%	18.11%	0.63%	7.67%	10.74%	29.37%	34.88%	9.61%	12.43%
	select(10 fds)	119.33%	10.24%	9.93%	10.25%	1.26%	3.00%	5.49%	15.05%	16.96%	4.59%	6.37%
	select(100 TCP fds)	1037.33%	59.03%	49.00%	~0%	~0%	~0%	5.08%	1.78%	9.29%	0.39%	7.43%
	fstat()	489.79%	15.31%	13.22%	7.91%	~0%	4.46%	12.92%	16.30%	26.68%	8.36%	14.64%
	mmap()/munmap()	180.88%	7.24%	6.62%	1.97%	1.12%	4.83%	5.89%	7.57%	8.71%	6.86%	8.27%
	fork()+exit()	208.86%	14.32%	14.26%	7.22%	~0%	12.37%	16.57%	24.03%	21.48%	13.77%	11.64%
	fork()+execve()	191.83%	10.30%	21.75%	23.15%	~0%	13.93%	16.38%	29.91%	34.18%	17.00%	17.42%
	fork()+/bin/sh	113.77%	11.62%	19.22%	12.98%	6.27%	12.37%	15.44%	23.66%	22.94%	18.40%	16.66%
	sigaction()	63.49%	0.19%	~0%	0.16%	1.01%	0.59%	2.20%	0.46%	2.27%	0.95%	2.43%
	Signal delivery	123.29%	18.05%	16.74%	7.81%	1.12%	3.49%	4.94%	11.39%	13.31%	5.37%	6.52%
	Protection fault	13.40%	1.26%	0.97%	1.33%	~0%	1.69%	3.27%	3.34%	5.73%	1.60%	3.39%
	Page fault	202.84%	~0%	~0%	7.38%	1.64%	7.83%	9.40%	15.69%	17.30%	10.80%	12.11%
	Pipe I/O	126.26%	22.91%	21.39%	15.12%	0.42%	4.30%	6.89%	19.39%	22.39%	6.07%	7.62%
	UNIX socket I/O	148.11%	12.39%	17.31%	11.69%	4.74%	7.34%	10.04%	16.09%	16.64%	6.88%	8.80%
	TCP socket I/O	171.93%	25.15%	20.85%	16.33%	1.91%	4.83%	8.30%	21.63%	24.43%	8.20%	9.71%
	UDP socket I/O	208.75%	25.71%	30.89%	16.96%	~0%	7.38%	12.76%	24.98%	26.80%	11.22%	13.28%
Bandwidth	Pipe I/O	46.70%	0.96%	1.62%	0.68%	~0%	0.59%	1.00%	2.80%	3.53%	0.78%	1.61%
	UNIX socket I/O	35.77%	3.54%	4.81%	6.43%	1.43%	2.79%	3.39%	5.71%	7.00%	3.17%	3.41%
	TCP socket I/O	53.96%	10.90%	10.25%	6.05%	~0%	3.71%	4.40%	9.82%	9.85%	3.64%	4.87%
	mmap() I/O	~0%	~0%	~0%	~0%	~0%	~0%	~0%	~0%	~0%	~0%	~0%
	File I/O	23.57%	~0%	~0%	0.67%	0.28%	1.21%	1.46%	1.81%	2.23%	1.74%	1.92%

Table 3.1: kR^X runtime overhead on the LMBench micro-benchmark (% over vanilla Linux; x86-64).

Table 3.2 summarizes our results on x86. The column `SEG` corresponds to the overhead of the R^X enforcement alone (i.e., `kR^X-KAS` and adjusted segment limits), whereas columns `SEG+D` and `SEG+X` correspond to the overhead of the full protection schemes, when using the return address decoys and return address encryption protection schemes respectively. The enforcement of `kR^X-SEG` incurs an overhead of up to 10.66% (avg. 0.33%) on latency and 2.46% (avg. 0.68%) on bandwidth. When coupled with fine-grained KASLR and the return addresses are protected using decoys, the overhead on latency is up to 16.22% (avg. 6.63%) and on bandwidth is up to 5.95% (avg. 2.57%), whereas when the return addresses are encrypted the overhead is slightly higher; up to 20.46% (avg. 8.98%) on latency and up to 5.23% (avg. 3.16%) on bandwidth. Note that we did not measure the overhead of fine-grained KASLR alone; since `kR^X-SEG` incurs negligible overhead, we expect performance to be similar to `SEG+D` and `SEG+X`.

In a nutshell, the impact of kR^X on I/O bandwidth ranges from negligible to moderate. As far as the latency is concerned, different kernel subsystems and services are affected dissimilarly; `open()/close()`, `read()/write()`, `fork()+execve()`, `select` (100 TCP fds), and pipe and socket I/O suffer the most.

3.6.2.2 Macro-benchmarks

To gain a better understanding of the performance implications of kR^X on realistic conditions, we used PTS [162]; PTS offers a number of *system* tests, such as `ApacheBench`, `DBench`, and `IOzone`, along with real-world workloads, like extracting and building the Linux kernel. Note that PTS executes each test at least five times but will execute it more times if the relative standard deviation is larger than a specific threshold (namely 3.5%). Table 3.3 presents the overhead for each benchmark on x86-64, under the different memory protection (`SFI`, `MPX`) and code diversification (`D`, `X`) schemes that kR^X provides. Similarly, Table 3.4 presents the overhead of the same benchmarks on x86 (i.e., the overhead of `SEG`, along with fine-grained KASLR, and both `D` and `X` schemes).

On x86-64, if the CPU lacks MPX support, the average overhead of full protection, across all benchmarks, is 4.04% (`SFI+D`) and 3.63% (`SFI+X`), respectively. When MPX support is available, the overhead drops to 2.32% (`MPX+D`) and 2.62% (`MPX+X`). The impact

	Benchmark	`SEG`	`SEG+D`	`SEG+X`
Latency	`syscall()`	0.47%	1.40%	1.33%
	`open()/close()`	~0%	12.26%	17.36%
	`read()/write()`	0.20%	6.29%	9.47%
	`select(`10 `fds)`	0.05%	5.80%	6.44%
	`select(`100 TCP `fds)`	~0%	9.89%	16.08%
	`fstat()`	1.11%	10.66%	12.69%
	`mmap()/munmap()`	~0%	6.25%	8.32%
	`fork()+exit()`	0.11%	6.06%	6.33%
	`fork()+execve()`	3.94%	12.93%	14.93%
	`fork()+/bin/sh`	~0%	7.24%	7.07%
	`sigaction()`	0.03%	1.02%	~0%
	Signal delivery	0.12%	4.92%	9.74%
	Protection fault	~0%	1.58%	4.10%
	Page fault	~0%	8.91%	11.27%
	Pipe I/O	~0%	~0%	1.80%
	UNIX socket I/O	~0%	~0%	~0%
	TCP socket I/O	~0%	16.22%	20.46%
	UDP socket I/O	~0%	7.92%	14.22%
Bandwidth	Pipe I/O	2.46%	5.95%	5.23%
	UNIX socket I/O	0.89%	2.09%	4.60%
	TCP socket I/O	~0%	2.65%	3.49%
	`mmap()` I/O	0.06%	~0%	0.04%
	File I/O	~0%	2.14%	2.45%

Table 3.2: kR^X runtime overhead on the LMBench micro-benchmark (% over vanilla Linux; x86).

of code diversification (i.e., fine-grained KASLR plus return address decoys or return address encryption) ranges between 0%–10% (0%–4% if we exclude `PostMark`). The `PostMark` benchmark exhibits the highest overhead, as it spends ∼83% of its time in kernel mode, mainly executing `read()/write()` and `open()/close()`, which according to Table 3.1 incur relatively high latency overheads. Lastly, it is interesting to note the interplay of `kR^X-{SFI, MPX}` with fine-grained KASLR, and each of the two return address protection methods (`D`, `X`). Although in both cases there is a performance difference between the

Benchmark	Metric	`SFI`	`MPX`	`SFI`+D	`SFI`+X	`MPX`+D	`MPX`+X
Apache	Req/s	0.54%	0.48%	0.97%	1.00%	0.81%	0.68%
PostgreSQL	Trans/s	3.36%	1.06%	6.15%	6.02%	3.45%	4.74%
Kbuild	sec	1.48%	0.03%	3.21%	3.50%	2.82%	3.52%
Kextract	sec	0.52%	~0%	~0%	~0%	~0%	~0%
GnuPG	sec	0.15%	~0%	0.15%	0.15%	~0%	~0%
OpenSSL	Sign/s	~0%	~0%	0.03%	~0%	0.01%	~0%
PyBench	msec	~0%	~0%	~0%	0.15%	~0%	~0%
PHPBench	Score	0.06%	~0%	0.03%	0.50%	0.66%	~0%
IOzone	MB/s	4.65%	~0%	8.96%	8.59%	3.25%	4.26%
DBench	MB/s	0.86%	~0%	4.98%	~0%	4.28%	3.54%
PostMark	Trans/s	13.51%	1.81%	19.99%	19.98%	10.09%	12.07%
Average		**2.15%**	**0.45%**	**4.04%**	**3.63%**	**2.32%**	**2.62%**

Table 3.3: kR^X runtime overhead on the Phoronix Test Suite (% over vanilla Linux; x86-64).

two approaches, for `SFI` this is in favor of `X` (encryption), while for `MPX` it is in favor of `D` (decoys).

In x86, the overhead of `kR^X-SEG` ranges from negligible to 4.62%, with an average of 0.77%, showcasing the efficiency of using the segmentation unit to enforce boundaries on memory operations (on real-world workloads). When coupled with fine-grained KASLR, and the return addresses are protected with decoys, the overhead is increased to a maximum of 6.13%, with an average of 1.32%, while with return address encryption the maximum overhead is 4.85% and the average is 1.69%. Note that, similarly to `MPX`, the overhead of encrypting the return addresses is (slightly) larger than employing return address decoys. This indicates that return address decoys are better suited for schemes that utilize hardware assistance, while return address encryption is more suitable for older CPUs that need to use the software-only `SFI` scheme to protect their kernels.

Benchmark	Metric	SEG	SEG+D	SEG+X
Apache	Req/s	0.20%	0.13%	0.21%
PostgreSQL	Trans/s	~0%	4.38%	5.29%
Kbuild	sec	0.27%	0.97%	1.57%
Kextract	sec	0.32%	1.13%	0.43%
GnuPG	sec	0.15%	0.15%	0.26%
OpenSSL	Sign/s	0.01%	0.01%	0.01%
PyBench	msec	0.14%	~0%	~0%
PHPBench	Score	~0%	0.20%	0.23%
IOzone	MB/s	~0%	1.41%	2.65%
DBench	MB/s	2.72%	0.07%	3.10%
PostMark	Trans/s	4.62%	6.13%	4.85%
Average		**0.77%**	**1.32%**	**1.69%**

Table 3.4: kR^X runtime overhead on the Phoronix Test Suite (% over vanilla Linux; x86).

3.6.3 Security

3.6.3.1 Direct ROP/JOP

To assess the effectiveness of kR^X against direct ROP/JOP attacks, we used the ROP exploit for CVE-2013-2094 [206], targeting Linux v3.8. We first verified that the exploit was successful on the appropriate kernel, and then tested it on the same kernel armed with kR^X. The exploit failed, as the ROP payload relied on pre-computed (gadget) addresses.

We then compared the vanilla and kR^X-armed `vmlinux` images. First, we dumped all functions and compared their addresses; under kR^X no function remained at its original location (function permutation). Second, we focused on the internal layout of each function separately, and compared them (vanilla vs. kR^X version) byte-by-byte; again, under kR^X no gadget remained at its original location (code block permutation). Recall that the default value (`k`) for the entropy of each routine is set to 30. Hence, even in the extreme scenario of a pre-computed ROP payload that uses gadgets only from a single routine, the probability of guessing their placement is $\mathtt{P_{succ}} = \mathtt{1}/\mathtt{2}^{\mathtt{30}}$, which we consider to be extremely low.

3.6.3.2 Direct JIT-ROP

As there are no publicly-available JIT-ROP exploits for the Linux kernel, we retrofitted an arbitrary read vulnerability in the `debugfs` pseudo-filesystem, reachable by user mode.[10] Next, we modified the previous exploit to abuse this vulnerability and disclose the locations of the required gadgets by reading the (randomized) kernel `.text` section. Armed with that information, the payload of the previously-failing exploit is adjusted accordingly. We first tested with fine-grained KASLR enabled, and the R^X enforcement disabled, to verify that JIT-ROP works as expected and indeed bypasses fine-grained randomization. Then, we enabled the R^X enforcement and tried the modified exploit again; the respective attempt failed as the code section (`.text`) cannot be read under R^X.

3.6.3.3 Indirect JIT-ROP

To launch an indirect JIT-ROP attack, code pointers (i.e., return addresses and function pointers) need to be harvested from the kernel's data region. Due to code block randomization, the knowledge of a return site cannot be used to infer the addresses of gadgets relative to the return site itself (the instructions following a return site are always placed in a permuted code block). Yet, an attacker can still leverage return sites to construct ROP payloads with `call`-preceded gadgets [23, 55, 86]. In kR^X, return addresses are either encrypted, and hence their leakage cannot convey any information regarding the placement of return sites, or "hidden" among decoy addresses, forcing the attacker to guess between two gadgets (i.e., the real one and the tripwire) for every `call`-preceded gadget used; if the payload consists of n such gadgets the probability of succeeding is $\mathtt{P}_{\mathtt{succ}} = \mathtt{1}/\mathtt{2}^{\mathtt{n}}$.

Regarding function pointers (i.e., addresses of function entry points that can be harvested from the stack, heap, or global data regions, including the interrupt vector table and system call table) or *leaked* return addresses (Section 3.7.1), due to function permutation, their leakage does not reveal anything about the immediate surrounding area of the disclosed routine. In addition, due to code block permutation, knowing any address of a function (e.g., either the starting address or a return site) is not enough for disclosing the

[10] The vulnerability allows an attacker to set (from user mode) an `unsigned long` pointer to an arbitrary address in kernel space, and read `sizeof(unsigned long)` bytes by dereferencing it.

exact addresses of gadgets within the body of this function. Recall that code block permutation inserts `jmp` instructions (for connecting the permuted basic blocks) both in the beginning of the function (to transfer control to the original entry block) and after every call site. As the per-routine entropy is at least 30 bits, the safest strategy for an attacker is to reuse whole functions. However, in both x86 and x86-64 Linux kernels, function arguments are passed in registers; specifically, the first 3 arguments on x86 and the first 6 arguments on x86-64 [20, 142]. This necessitates the use of gadgets for loading registers with the proper values. In essence, kR^X effectively restricts the attacker to data-only type of attacks on function pointers [176] (e.g., overwriting function pointers with the addresses of functions of the same, or lower, arity [61]).

3.7 Discussion

3.7.1 Limitations

3.7.1.1 Substitution Attacks

Both return address protections are subject to *substitution attacks*. To illustrate the main idea behind them, we will be using the return address encryption scheme (return address decoys are also susceptible to such attacks). Assume two call sites for function `f`, namely CS$_1$ and CS$_2$, with RS$_1$ and RS$_2$ being the corresponding return sites. If `f` is invoked from CS$_1$, RS$_1$ will be stored (encrypted) in a kernel stack as follows: $\texttt{[RS}_1\texttt{\^{}xkey}_\texttt{f}\texttt{]}$. Likewise, if `f` is invoked from CS$_2$, RS$_2$ will be saved as $\texttt{[RS}_2\texttt{\^{}xkey}_\texttt{f}\texttt{]}$. Hence, if an attacker manages to leak both "ciphertexts," though they cannot recover RS$_1$, RS$_2$, or $\texttt{xkey}_\texttt{f}$, they may replace $\texttt{[RS}_1\texttt{\^{}xkey}_\texttt{f}\texttt{]}$ with $\texttt{[RS}_2\texttt{\^{}xkey}_\texttt{f}\texttt{]}$ (or vice versa), thereby forcing `f` to return to RS$_2$ when invoked from CS$_1$ (or to RS$_1$ when invoked from CS$_2$). Note that replacing $\texttt{[RS}_1\texttt{\^{}xkey}_\texttt{f}\texttt{]}$, or $\texttt{[RS}_2\texttt{\^{}xkey}_\texttt{f}\texttt{]}$, with *any* harvested (encrypted) return address, say $\texttt{[RS}_n\texttt{\^{}xkey}_\texttt{f'}\texttt{]}$, is not a viable strategy because the respective return sites (RS$_1$/RS$_2$, RS$_n$) are encrypted with different keys ($\texttt{xkey}_\texttt{f}$, $\texttt{xkey}_\texttt{f'}$)—under return address encryption (`X`), substitution attacks are only possible among return addresses encrypted with the same `xkey`.

Substitution attacks resemble the techniques for overcoming coarse-grained CFI by stitching together `call`-preceded gadgets [23, 55, 86]. However, in such CFI bypasses, *any* `call`-preceded gadget can be used as part of a code-reuse payload, whereas in a substitution attack, for every function `f`, the (hijacked) control flow can only be redirected to the *valid* return sites of `f`, and, in particular, to the subset of those valid sites that can be leaked *dynamically* (i.e., at runtime). Leaving aside the fact that the number of `call`-preceded gadgets, at the attacker's disposal, is highly limited in such scenarios, both our return address protection schemes aim at thwarting JIT-ROP, and, therefore, are not geared towards ensuring the integrity of code pointers [124].

3.7.1.2 Race Hazards

Both schemes presented in Section 3.4.5.4 obfuscate return addresses *after* they have been pushed (in cleartext) in the stack. Although this approach entails changes only at the callee side, it does leave a window open for an attacker to probe the stack and leak unencrypted/real return addresses [31]. Chapter 4 describes a different system (kSplitStack) to protect return addresses that does not suffer from those limitations, based on relocating return addresses to the protected region.

3.7.2 Handling Violations

As mentioned in Section 3.4, when kR^X detects an attempt to read the protected region, it calls `krx_handler` which logs debugging information to the kernel log and halts the system. Nonetheless, kR^X can be configured to employ custom violation handlers using the `-fplugin-arg-krx-stub` knob. To facilitate user needs, the RCs implementation could be slightly adjusted to pass different information to the handler.

In the current implementation of `kR^X-SFI`, we add one call to `krx_handler` in every function and redirect all RCs to it. This allows the handler to log the function that the violation occurred before halting the system. Snippet 3.1 shows how the injected instrumentation should be modified to facilitate users that would like to obtain the specific address/RC which triggered the violation.

This RC changes from a "never-taken" to an "always-taken" branch, since for every memory read that does not target the code section, the branch is taken. Note that this instrumentation: *(a)* is compatible with all optimizations (`O0`–`O3`) we discussed, and *(b)* al-

```
pushf
lea 0x154(%rsi),%r11
cmp $_krx_edata,%r11
jbe safe_lbl
call krx_handler
safe_lbl: popf
```

Listing 3.1: Alternative RC instrumentation applied on a memory read

lows the kernel to continue its execution once the handler is executed (i.e., if the handler does not halt the system). Admittedly, we employed this version of the instrumentation during the development of kR^X and found it very useful for debugging purposes, however it increases the memory footprint of the kR^X instrumentation and our internal measurements indicate that it induces a slightly higher overhead than the "never-taken" approach which makes it less suitable for everyday use.

Chapter 4

kSplitStack

4.1 Overview

Protecting code pointers is fundamental in order to prevent code reuse attacks as discussed in Section 2. Unfortunately, many code pointer protection schemes (including the ones presented in Section 3.4) suffer from race hazards: an adversary can leak or corrupt them before they are protected by constantly probing the address that it resides, therefore severely undermining the effectiveness of the deployed scheme.

In this chapter we discuss the practicality and effectiveness of such attacks and investigate whether a defense solution (*kSplitStack*) that does not suffer from this weakness can efficiently and effectively protect OS kernels.

4.1.1 Threat Model

Adversarial Capabilities We assume an *unprivileged* local attacker (i.e., with the ability to execute, or control the execution of, user programs on the OS) who seek to execute arbitrary code with elevated privileges by exploiting kernel-memory corruption bugs. Specifically, the attacker is armed with an *arbitrary memory disclosure* bug [204, 207] which that may be triggered *multiple* times, thereby leaking *any* kernel memory address. Additionally, she also controls an *arbitrary memory corruption* bug [165, 203, 205, 208–211] that allows her to corrupt the contents of *any* kernel-space memory address. Finally, the attacker is able to trigger *hardware events* (e.g., interrupts, exceptions) [101] at will, without halting or otherwise impeding the execution of the kernel. Microarchitectural attacks, like Meltdown [134], Spectre [114], RIDL [188] and similar side-channel attacks [91], are considered out of scope.

Hardening Assumptions We presume an OS that is not vulnerable to direct code injection attacks by enforcing the W^X policy [125, 132, 185] in kernel space. We also assume that the kernel is hardened against ret2usr attacks using either hardware (e.g., Intel SMEP [200] and SMAP [39], ARM PXN [3] and PAN [18]) or software (e.g., KERNEXEC [156], UDEREF [154, 155], kGuard [109], KPTI [43]) solutions. Additionally, we assume that the kernel is hardened against function pointer tampering [168, 170] using a function pointer protection scheme such as CFI [50, 79, 97, 129, 146, 183]. We assume sane (read-only) memory permissions for the Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT) [32, 76]. We also consider a kernel that is not vulnerable to page table tampering [127] by self-protecting the page tables [52, 53]. Finally, the kernel may have support for kernel-space ASLR [59], stack-smashing protection [186], proper `.rodata` sections (constification of critical data structures) [185], pointer (symbol) hiding [164] or any other kernel hardening feature —they are orthogonal to kSplitStack. Data-only attacks such as credentials modification [198] are considered out of scope and their protection [26] is also orthogonal to kSplitStack.

Overall, the adversarial capabilities of the attacker in kSplitStack are realistic and resemble the capabilities assumed by code reuse defenses that protect user space applications [5, 6, 17, 48, 49, 54, 84, 174, 187].

4.2 Effectiveness of Race Hazards

There are two main ways that architectures facilitate storing return addresses when a subroutine is called: employing a register or storing them to the program stack. The former is typically preferred by RISC architectures with many registers (e.g., ARM [3], Power ISA [105]) and even though return addresses need to be stored in memory in the case of nested calls, it can be beneficial for leaf functions. The latter approach, on the other hand, is typically employed by CISC architectures with a limited number of registers (e.g., x86/x86-64 [104]) and delegates storing return addresses in the stack to the hardware.

Many return address protection schemes that target the x86/x86-64 architecture protect return addresses *after* the hardware emits them to the stack [21, 51, 89, 157], similarly to the return address protection schemes we present in Section 3.4. Unfortunately, this renders them vulnerable to race conditions; an attacker could leak or corrupt a return address in the stack *before* it is protected. On legacy systems, such attacks relied on multithreaded scheduling in order to access the victim stack before the return address is protected. However, on modern systems with multiple CPU cores, this problem is exacerbated since attackers can "pin" their attack program on one core and probe vulnerable programs that execute on different cores at real time [31].

Even though this methodology has been well known, exploiting such vulnerabilities in a reliable fashion has been seen as extremely hard due to the narrow race windows, as evidenced by the discussions of the authors of almost all the aforementioned vulnerable systems. To assess the effectiveness of this methodology in the kernel setting, we perform the following experiment: in a kR^X-protected kernel using the return address encryption scheme we spawn a victim and an attacker process, "pinning" them to different CPU cores. The victim process repeatedly calls the `read` system call, reading a large 100MB file, while the attacker process repeatedly uses the arbitrary memory read vulnerability described in Section 3.6.3 until it wins the race (i.e., obtains the plaintext return addresses). Note that since kR^X, similarly to the rest of the vulnerable systems, does not change the stack layout —e.g., by adding or removing local variables— an attacker can always know *a priori* the exact offsets in the victim stack that hold the return address and as per the threat model of kR^X an attacker could repeat this procedure for all system calls.

We ran the experiment ten times and measured the time it took the attacker process to leak the first three return addresses pushed in the stack. On average, it took the attacker process ∼4 milliseconds to leak all three return addresses with ∼1443 read attempts. These results indicate that even in the (more challenging) kernel setting[1], race conditions is a

[1] Kernel attackers need to employ the system call interface in order to exploit the arbitrary memory read vulnerability, whereas in user space one thread can freely read the other thread stack.

severe weakness that can completely undermine the security of defense solutions. This is further evidenced by the decision of Microsoft to stop the deployment of Return Flow Guard (RFG) due to a race conditions vulnerability discovered by their red team [10], as well as past research that employed race conditions to bypass other CFI implementations [31].

4.3 Approach

The current stature of defenses against x86/x86-64 kernel code reuse attacks is vulnerable against race conditions targeting code pointers (more specifically return addresses). In the previous section (Section 4.2) we show that the return address protection schemes of kR^X can be easily bypassed by exploiting the race hazards they suffer from. IskiOS [89] incorporates a shadow stack in the Linux kernel which is isolated through Intel Memory Protection Keys (MPK) [94]. While the use of MPK for user space memory isolation is already shown [184], IskiOS takes advantage of the introduction of KPTI [43] to safely employ MPK in the kernel setting. Unfortunately, similarly to kR^X, it is vulnerable against race conditions both when copying the return address to the protected shadow stack in the function prologue and when copying it back to the stack in the function epilogue.

Implementations of shadow stacks tailored to user space are not impeccable either. Some state-of-the-art implementations [21, 51] suffer from the same weakness (introduction of race conditions) and are therefore not suitable solutions for our threat model. The original CFI paper [2], avoids race conditions by replacing the `call/ret` pair with indirect `jmp` instructions. Unfortunately this approach has been shown to impose significant overhead [51], probably because the target prediction mechanisms of indirect jumps is not as effective as the ones employed when returning from a function call [197].

ASLR-Guard [136] and Code Pointer Integrity [124] also introduce shadow stack implementations that do not suffer from race conditions. They reserve a general-purpose register and employ it as the stack pointer of the unprotected stack and repurpose `%rsp` to point to the shadow stack. Then they modify the compiler to emit instructions that modify the reserved register when storing local variables or spilling registers. This scheme is not only safe from race hazards but it also avoids the costly conversion of `call/ret` instructions to indirect jumps. Unfortunately, it is not an ideal fit for the kernel setting where a large percentage of the code base is low-level handwritten assembly code; every explicit `%rsp`-based instruction needs to be carefully rewritten so that it uses the appropriate register (potentially with a different offset since under these schemes the return address is not present in the stack frame) while instructions that push and pop values to the stack could lead to increased overhead since they would have to be replaced with `sub/mov` and `mov/add` sequences respectively. More importantly, instructions that implicitly use the `%rsp` register might be impossible to be rewritten in a manner that does not utilize the protected stack. As an example, the kernel often spills/fills the value of the `%rflags` register when it disables interrupts (e.g., before entering a spinlock). This is achieved with the `pushf/popf` instructions which implicitly employ the `%rsp` register. Such instances showcase that it is infeasible to avoid storing non-sensitive values in the shadow stacks, despite the authors intent.

To fill the gap of a race-free, kernel-tailored shadow stack we present kSplitStack: a novel scheme that is based on relocating the stack pointer to an isolated region `before` any function call —therefore keeping the return address always protected— while also forcing the hardware to emit code pointers (e.g., control data in the interrupt context) in the protected region thus protecting another weakness of current state-of-the-art kernel defenses.

4.4 Design

4.4.1 kSplitStack Region

User space processes and threads (from this point on we will refer collectively to both as *tasks*), in addition to their user space stacks, also have their own dedicated kernel stacks. These stacks are used by kernel code whenever a task performs a system call or whenever the execution of a task triggers an exception (e.g., a page fault). Unfortunately, even though return addresses are becoming a valuable target for attackers, the kernel provides no protection against their leakage [31, 54] or corruption [12]. kSplitStack provides strong protection against any unauthorized return address access in the kernel stack, by relocating all return addresses to an isolated region, in an approach similar to a shadow stack. In

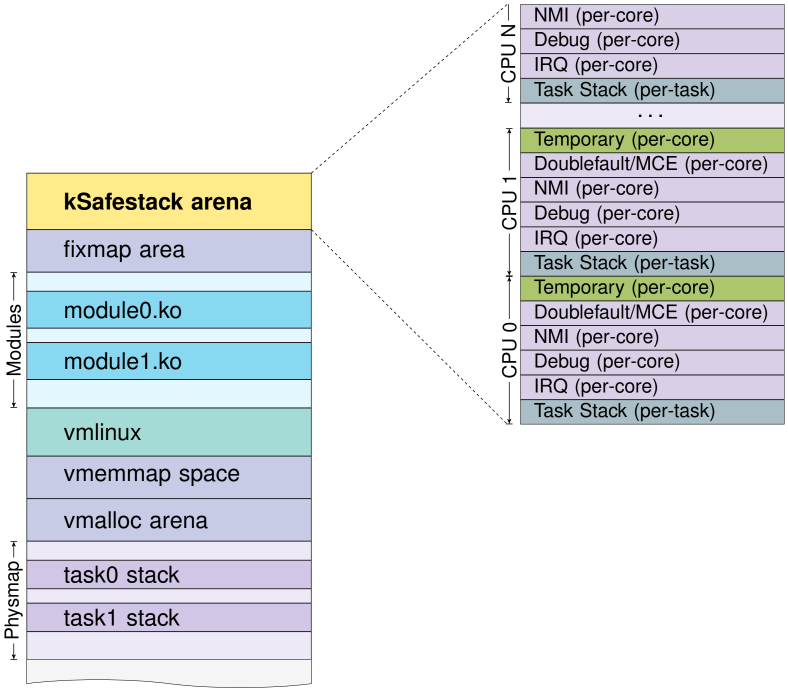


Figure 4.1: The high level overview of kSplitStack.

contrast to previous user space shadow stack implementations kSplitStack is tailored to the kernel setting, which as we will explain in the following sections poses unique challenges which require special consideration and handling.

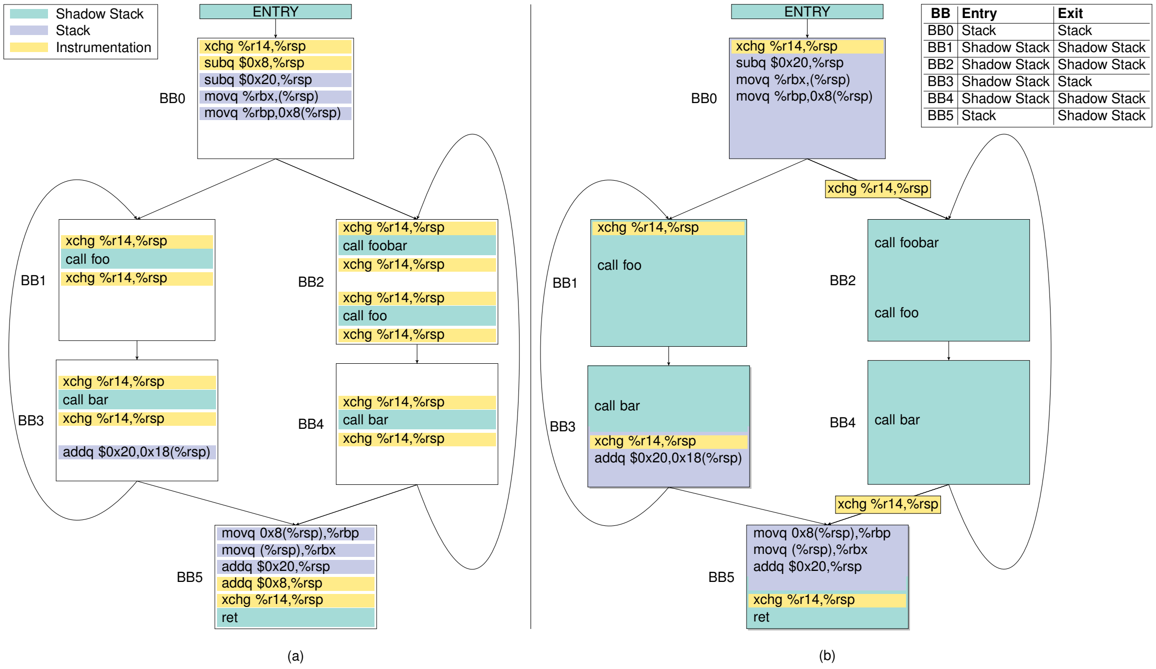
Every time a task is created, kSplitStack reserves additional physical page frames and employs them as the physical memory of the task shadow stack. Whenever this process is scheduled for execution, these page frames are mapped in the kSplitStack arena: an isolated region on the top of the address space, protected by range checks (Figure 4.1). Carving this region from the top of the address space facilitates its efficient protection: if a parallel shadow stack scheme was employed then kSplitStack would have to employ multiple range checks —one for every shadow stack— which, in turn, would result to excessive performance overhead. On the other hand, if kSplitStack carved the region in a location surrounded by other data, it would require checking both its bounds, whereas by placing it at a completely disjoint region on the top, kSplitStack only needs to check its lower bound.

Figure 4.1 shows the internal structure of the isolated region. The region is split into subregions, each used exclusively by a single CPU core. This allows kSplitStack to seamlessly protect kernel return addresses in modern, multicore systems where multiple tasks are executed simultaneously on different cores. Each subregion is further divided into six slots. The first slot corresponds to the currently executing task in the CPU, while the rest are related to hardware event handling (we discuss event handling in detail in Section 4.4.3). Every time a task is scheduled for execution, kSplitStack first identifies the core it will be executed and maps the reserved page frames of this task in the appropriate slot. It also invalidates the stale TLB entries of this slot, thus ensuring that each task can only access its own shadow stack. Note that kSplitStack supports the same number of CPUs as the kernel, since the size of the isolated region is determined by the `CONFIG_NR_CPUS` configuration option. To avoid physical memory waste when the actual number of CPUs in a system is smaller than the one specified by the option, the physical page frames that belong to the unused portion of the region are freed after boot.

Finally, even though the shadow stacks in the isolated region are protected against any unauthorized access, attackers might try to take advantage of the directly mapped memory (physmap). The physmap is a region in the kernel portion of the address space that maps all physical page frames to facilitate efficient dynamic memory allocation [108]. Without special consideration, the page frames used as shadow stacks would also be mapped in the (unprotected) physmap, where they could leaked or modified. kSplitStack meticulously unmaps the shadow stack page frames from the physmap whenever a task is created and zaps their contents before remapping them on task exit; therefore preventing attackers from accessing their aliases.

4.4.2 Relocating Return Addresses

Once the shadow stack page frames are mapped, kernel code can start using it to safely store return addresses. Unfortunately, code uses the stack to also store local variables thus simply redirecting the stack pointer to the shadow stack is not a viable option. Instead, kSplitStack injects lightweight instrumentation to ensure that only return addresses are emitted to the isolated region while local variables remain in the (unprotected) kernel stack. To achieve this, kSplitStack forces only `call` and `ret` instructions to use the shadow stack, while every



(a)

```
BB0
xchg %r14,%rsp
subq $0x8,%rsp
subq $0x20,%rsp
movq %rbx,(%rsp)
movq %rbp,0x8(%rsp)

BB1
xchg %r14,%rsp
call foo
xchg %r14,%rsp

BB2
xchg %r14,%rsp
call foobar
xchg %r14,%rsp

xchg %r14,%rsp
call foo
xchg %r14,%rsp

BB3
xchg %r14,%rsp
call bar
xchg %r14,%rsp

addq $0x20,0x18(%rsp)

BB4
xchg %r14,%rsp
call bar
xchg %r14,%rsp

BB5
movq 0x8(%rsp),%rbp
movq (%rsp),%rbx
addq $0x20,%rsp
addq $0x8,%rsp
xchg %r14,%rsp
ret
```

(b)

```
BB0
xchg %r14,%rsp
subq $0x20,%rsp
movq %rbx,(%rsp)
movq %rbp,0x8(%rsp)

xchg %r14,%rsp

BB1
xchg %r14,%rsp
call foo

BB2
call foobar

call foo

BB3
call bar

xchg %r14,%rsp
addq $0x20,0x18(%rsp)

BB4
call bar

xchg %r14,%rsp

BB5
movq 0x8(%rsp),%rbp
movq (%rsp),%rbx
addq $0x20,%rsp

xchg %r14,%rsp
ret
```

BB	Entry	Exit
BB0	Stack	Stack
BB1	Shadow Stack	Stack
BB2	Shadow Stack	Shadow Stack
BB3	Shadow Stack	Stack
BB4	Shadow Stack	Shadow Stack
BB5	Stack	Shadow Stack

Figure 4.2: kSplitStack instrumentation: (a) assembly code and (b) compiled code

other stack-based operation uses the kernel stack. Alas, both instructions are hardwired to implicitly employ the stack pointer (`%rsp`, Section 4.3) in their operations which, in turn, necessitates toggling the stack pointer value. Specifically, kSplitStack first reserves a general purpose register (in our prototype we reserved `%r14`[2]) which acts as the shadow stack pointer. It then uses the `xchg %r14,%rsp`[3] instruction to toggle the value of `%rsp` between the kernel stack and the shadow stack.

Due to the low-level nature of its operations, the kernel consists of both assembly code and compiled (C) code. kSplitStack meticulously instruments the functions in both sets of code to ensure that all return addresses are safely stored in the shadow stack, however the injected instrumentation differs. Specifically, in compiled code kSplitStack takes advantage of the intraprocedural (CFG) produced by the compiler to minimize the number of emitted instructions and optimize their placement, while it takes a more conservative approach in assembly code since its CFG is not available. We will be using the example function in

[2] We selected `%r14` because it is the callee-saved register used the least by handwritten assembly code.

[3] In x86-64, the `xchg` instruction swaps the values of its operands.

Figure 4.2 to describe the two flavors of kSplitStack instrumentation on the same function. Note that in assembly code the CFG is not available to kSplitStack, however we employ the same depiction for both instrumentations for graphical consistency.

Figure 4.2(a) illustrates the emitted instrumentation on assembly code. Every function call emits its return address to the isolated region, therefore the `%rsp` at the function entry always points to the shadow stack. kSplitStack emits an `xchg` instruction to toggle its value in order to allow the rest of the function body to use the kernel stack for its local variables. It then “emulates” pushing a return address in the kernel stack by subtracting eight bytes from `%rsp`. This is necessary because in the (unlikely) case that the function takes more than six arguments, additional arguments are passed through the stack [142]. Since these arguments are placed in the caller stack frame, the instructions that access them contain offsets computed with the assumption that the return address is pushed in the kernel stack. As a result, by “emulating” this operation kSplitStack ensures that these offsets are correct.

kSplitStack then identifies all `call` instructions in the function code stream and surrounds them with `xchg` instructions. The `xchg` that precedes the `call` ensures that the emitted return address is placed in the shadow stack, while the one that succeeds it switches it back to the kernel stack to facilitate the correct use of local variables in subsequent instructions (e.g., as in the case of BB3). Finally, in the function epilogue kSplitStack “emulates” poping the return address from the kernel stack by adding eight bytes to `%rsp` before toggling its value to the shadow stack to facilitate the use of the real return address by the `ret` instruction.

Figure 4.2(b) shows the instrumentation of kSplitStack on compiled code, where access to the CFG is available. The first step is to eliminate the need for the `sub`/`add` pair of the assembly instrumentation. kSplitStack achieves this by tracking all stack operations and maintaining the depth of the stack at any given instruction. Then it examines every stack-based access and compares its offset with the computed depth; if it is larger, it denotes an attempt to access an argument from the stack and kSplitStack adjusts its offset.

Regarding the toggling of the `%rsp` value, kSplitStack follows a four step approach aimed at optimizing both their placement and number:

1. In this bookkeeping step, kSplitStack examines the instructions of each basic block

looking for stack-based operations. If the first stack-based operation in a basic block uses the shadow stack (e.g., a `call` instruction), then kSplitStack sets the entry state of this basic block as shadow stack. On the other hand, if the first stack-based operation uses the kernel stack, then kSplitStack sets the basic block entry state as stack. Similarly, depending on the last stack-based operation of the basic block, kSplitStack determines its exit state.

2. kSplitStack scans each basic block instruction and toggles the value of `%rsp` whenever there are instructions within the basic block that use different stacks. In Figure 4.2(b), the `xchg` instruction in BB3 would be injected at this step, since kSplitStack would identify that the `addq` instruction references a local variable in the kernel stack after the `call` instruction which uses the shadow stack. Similarly in BB5 the `ret` uses the shadow stack while the previous stack-based operations use the kernel stack.

3. kSplitStack compares the entry state of each basic block with the exit state of its predecessors. If the exit state of *all* the predecessors is different than the entry state of the basic block, then it emits an `xchg` instruction in the beginning of the basic block. The `xchg` instruction in BB1 is emitted at this stage, since the exit state of both predecessors (BB0 and BB3) is stack while the entry state of BB1 is shadow stack. Additionally, since the entry state of BB0 is stack and the exit state of the (dummy) entry basic block is shadow stack, kSplitStack emits the `xchg` instruction in the beginning of BB0.

4. For every remaining basic block, kSplitStack examines the exit state of its predecessors and if different than the basic block entry state it adds `xchg` instructions on the *edges* that connect them. This facilitates supporting basic blocks with predecessors that have mixed exit states without unnecessarily toggling the value of `%rsp`. The `xchg` instruction on the edges that connects BB0 with BB2 is emitted at this step, as well as on the edge that connects BB4 with BB5.

In the example of Figure 4.2, the compiled code flavor of the instrumentation halved the number of emitted `xchg` instructions (six from twelve) compared to the assembly code flavor, while also completely eliminating the instrumentation in the BB2-BB4 loop. In our testbed (Section 4.6), this flavor results in a 31% reduction of the emitted toggling instructions.

The proposed instrumentation scheme offers a number of significant benefits to kSplitStack. Firstly, keeping the shadow stack pointer to a reserved register simplifies its protection, since no instruction can overwrite it and it is never spilled to unprotected memory. Additionally, toggling the value of `%rsp` alleviates the need to explicitly update its value, since it is modified automatically by the `call` and `ret` instructions. Most importantly, it facilitates *race-free* protection of return addresses, since return addresses are always placed in the isolated region and are never exposed in unprotected memory.

4.4.3 Handling Hardware Events

One of the most important aspects of kernel software is dealing with synchronous and asynchronous hardware events. Synchronous events (or exceptions) are triggered when —user space or kernel— code performs some operation that the CPU is unable to handle. For instance, accessing a memory address that is not mapped in the page tables will result in a page fault. Asynchronous events (or interrupts) on the other hand are triggered at random times, as a response to hardware signals (e.g., by incoming network traffic). Whenever an event of either category occurs, the control flow is paused, the processor state at the time of the event is stored —including the value of of the instruction pointer (`%rip`) at the time of the event— and special kernel code is executed to handle it. Once the event is handled, the stored value of `%rip` is used as a kernel return address to facilitate resuming the paused control flow. In this section we discuss in detail the challenges of protecting this special type of return address and how kSplitStack protects it in a secure and practical manner.

Figure 4.3 shows the processor state stored when handling an event on a vanilla x86-64 Linux kernel. The top five entries are automatically pushed in the stack by the hardware when the event is triggered, while the rest are spilled by low-level assembly code. The kernel takes advantage of the ABI [142] and avoids spilling the callee-saved registers unless necessary (i.e., the handler of this event might need to access them), since their value will remain unmodified throughout the execution of compiled code.

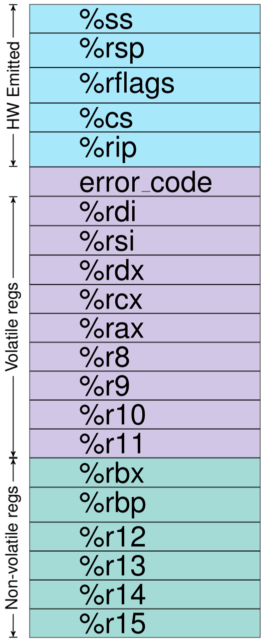


Figure 4.3: The processor state stored when an event handler is executing.

This information is spilled in the stack that the event handler uses during its execution. Most exceptions are handled in the same stack as the one employed by the code that triggered the event, however interrupts and exceptions triggered in serious (potentially unrecoverable) situations migrate to different stacks to ensure that they are handled in known-good memory locations. Specifically, the events that migrate to a different stack are Interrupt Requests (IRQ), Non-Maskable Interrupts (NMI), Doublefault Exceptions, Machine Check Exceptions (MCE) and Debugging Exceptions.

In x86-64 systems, most events that require stack migration employ a hardware feature called *Interrupt Stack Table* (IST). IST is a single-dimensional (per-cpu) table which can be filled by the kernel with entries that point to the top of a stack. When the kernel registers the handler of events it also specifies the entry of the IST that should be utilized. If no such entry is specified, then the handler is executed without migrating to a different stack. Note that if an entry is specified, the IST first migrates to the appropriate stack and *then* allows the hardware to emit the first five entries of the processor state. The only type of event that does not utilize the IST to migrate to a different stack is IRQ, which instead performs the migration through the software. This happens to facilitate handling *nested* interrupts, where additional interrupts are triggered before the handler has finished its processing. Should the IST be used, the new processor state would be emitted at the same location (the top of the stack) effectively overwriting the previous one.

Similarly to the shadow stack instrumentation, kSplitStack protects this type of return addresses by ensuring that they are *never* exposed to unprotected memory. The main intuition behind the protection of kSplitStack lies on forcing the hardware to *always* spill the processor state in the protected region by taking advantage of the IST. As mentioned above, the IST ensures that the portion of the processor state that is emitted by the hardware is spilled *after* the stack migration. kSplitStack utilizes this observation to redirect it to the protected region and modifies the low-level code in the event handlers to also spill the register values there, thus retaining the processor state always secure.

kSplitStack first statically allocates in the protected region enough memory to hold the corresponding shadow stacks of the additional stacks for every CPU and unmaps their aliases from the physmap. It then modifies the IST entries to point to the top of the shadow stack slots rather than the top of the stacks for every event that migrates to a different stack. To reduce its memory footprint, kSplitStack coalesces the two shadow stacks of the exceptions (Doublefault and MCE) since these serious events typically trap the kernel and therefore their handlers are not contentious. Note that since these shadow stack slots correspond to the per-cpu additional stacks, they do not require updates on context switching; they are always mapped to the same page frames throughout the system execution.

Protecting the processor state in events that originally do not migrate to a different stack requires a slightly different approach. kSplitStack adds one more slot to each CPU representation in the protected region, the *temporary slot*, adds it to an IST entry and forces all (originally non-migrating) events to use it. kSplitStack then amends the low-level entry point of these event handlers to *copy* the hardware emitted processor state to the appropriate *shadow stack* location —either the kernel shadow stack or the IRQ shadow stack in the case of an IRQ— thus always retaining them in the protected region. Once copied, the registers are then spilled in the shadow stack therefore completing the processor state.

The last challenge kSplitStack has to address is allowing legitimate accesses to the processor state by event handlers. Some event handlers require accessing the state during their execution (e.g., if a page fault occurs in kernel memory, the handler examines the value of `%rip` to determine the handling of the event), however this is not allowed since it is now placed in the protected region. To facilitate this process, kSplitStack adds an additional compiler pass which detects all processor state pointers and replaces their dereferences with calls to special *getter and setter* functions, depending on whether the dereference is a memory read or write. These functions are exempt from the instrumentation and are therefore able to access the processor state. Note that kSplitStack does not assign them to any function pointers, thus they cannot be leaked to attackers.

4.5 Implementation

4.5.1 Isolation Enforcement

There are multiple ways to preserve the integrity and secrecy of the kSplitStack region, as discussed in Section 2. In our prototype implementation we adopted the enforcement mechanism of kR^X. However, since the kR^X mechanism only enforces the secrecy primitive on its protected region, we *augmented* it to also instrument memory writes; thus completely isolating the region[4].

4.5.2 Kernel Modifications

For our prototype we utilized the kernel patches provided by kR^X, which perform all the necessary kernel modifications to create an isolated region on the top of the address space. We modified these kernel patches to: *(a)* statically allocate enough virtual memory for the kSplitStack region, which is placed adjacently to the kR^X protected region (kernel image and modules `.text` sections), *b* map the appropriate physical page frames in the corresponding kSplitStack page table entries and flush stale TLB entries, *(c)* modify the

[4] We carefully engulfed our memory write instrumentation in the kR^X code so that optimizations would be applied on an access level instead of treating reads and writes separately, thereby obtaining maximal performance benefits (Section 4.6).

IST and all hardware event handlers to employ the corresponding stacks in the protected region, and *(d)* rewrite any handwritten assembly functions to avoid using `%r14`.

4.5.3 kSplitStack Instrumentation

Similarly to the isolation enforcement instrumentation, kSplitStack hooks additional passes to the pass manager at the intermediate representation (IR) level to inject its instrumentation. Specifically, our prototype inserts two passes: one that injects the return address relocation instrumentation and another that substitutes dereferences of control state pointers with calls to *accessor* function (see Section 4.4). We added the former at the latest stage of the RTL optimization phase to ensure its *precise* placement: both for correctness and for security reasons it is imperative that the inserted `xchg` instructions remain in the intended basic blocks and not moved to different ones. By adding the pass late in the compilation process, kSplitStack ensures that this condition is met. The latter pass is added early in the GIMPLE IR optimization phase for two reasons: *(a)* kSplitStack needs type information in order to detect such dereferences which are available in the early compilation stages but not in subsequent ones, and *(b)* by replacing the dereferences with calls as early as possible, we allow subsequent compiler optimization passes to optimize the produced binary and therefore reap performance benefits.

Since handwritten assembly code is not processed by the compiler, their return addresses would be exposed corruption and disclosure attempts by adversaries. To prevent this issue, kSplitStack extends its instrumentation to also relocate return addresses of assembly code. To instrument this code with the (unoptimized) return address relocation scheme, we added *assembler wrappers* which detect patterns that denote the prologue and epilogue of functions as well as any `call` instructions and instrument them accordingly.

4.5.4 Code Diversification

kSplitStack relies on code diversification to thwart direct code reuse attacks. kR^X its isolation mechanism with a fine-grained KASLR component which randomized the kernel code layout in order to probabilistically break such exploits (Section 3.4. However, because kR^X did not provide comprehensive protection of return addresses (especially in the case of return address decoys), it employed both inter- and intra-function diversification (function and code block permutation respectively).

kSplitStack provides significantly stronger return address protection thus intra-function diversification is not necessary. As a result, we couple kSplitStack with a different component based on Code Pointer Hiding (CPH) [48]. Specifically, we created a forward-edge "trampoline" for every function and replaced all function pointers with the appropriate "trampoline". This is a two step process: first an assembler wrapper creates the trampolines of exported functions and then the relocation information of the kernel binary (`vmlinux`) and the kernel modules are modified to ensure that all function pointers utilize the appropriate function pointers. In contrast to the original CPH, we did not create backward-edge trampolines or replaced the target of direct function calls with trampolines, since kSplitStack precludes return address leaks or corruptions. Finally, all functions and "trampolines" are permuted to ensure that their placement is randomized (inter-function diversification).

To assess the benefits of this approach, we also coupled kSplitStack with a fine-grained KASLR component, similar to the one of kR^X. This component does not protect return addresses but performs the rest of the diversifications (function and code block permutation). We compare the performance of the two schemes in the following section.

4.6 Evaluation

In this section we assess the performance impact of our kSplitStack implementation on the Linux kernel. To this end, we employ the LMBench suite [141] to perform microbenchmarks on various operations and services of the operating system. Additionally, we employ the Phoronix Test Suite (PTS) [162] to measure the imposed overhead on real-world applications. The reported results are averages of ten and five runs respectively. Measurements that involve code randomization (i.e., fine-grained KASLR or CPH) are the average of ten distinct measurements, each after a kernel recompilation. We focus on measuring the performance impact of kSplitStack on CPUs that support MPX. Additional measurements and discussion for CPUs that lack MPX support can be found in Appendix B.

4.6.1 Testbed

Our experiments were performed on a Debian GNU/Linux v7 system, with a 4GHz quad-core Intel Core i7-6700K (Skylake) CPU and 16GB of RAM. In all experiments the kernel (v3.19) was built with GCC v4.7.2 (which was also used to build the GCC plugins), with the default Debian configuration. Finally, the kernels were linked and assembled using binutils v2.25.

4.6.2 Performance Evaluation

4.6.2.1 Micro-benchmarks

For our first set of experiments we employed the LMBench [141] suite. LMBench measures the *latency* and *bandwidth* of various system calls and kernel operations in order to assess the performance of kSplitStack, therefore providing valuable fine-grained insight on its impact to specific kernel subsystems. Specifically, we focus on the latency of user to kernel and kernel to user context switch (`syscall()`) and of multiple commonly used system calls (`open()/close()`, `read()/write()`, `select()`, `fstat()`, `mmap()/munmap()`). In addition, we measured the latency of creating a process (`fork()+{exit()`, `execve()`, `/bin/sh}`), installing a signal handler (`sigaction()`) and delivery of a signal, handling page and protection faults, along with interacting with pipes and sockets (both UNIX and TCP/UDP). Finally, we also measured the impact on the bandwidth of pipe, socket and file I/O operations.

Table 4.1 summarizes our results. The second column (`W`) corresponds to the overhead of instrumenting with MPX range checks only memory write operations, the third (`RW`) both memory read and memory write operations, the fourth (`SS`) the overhead of employing a shadow stack (i.e., reserving the region on the top of the address space, performing all necessary page table modifications on context switch, relocating all return addresses to the region using the instrumentation described in Section 4.4) *without* isolating the kSplitStack region. The fifth (`RW+SS`) is the combination of `RW` and `SS`, therefore it illustrates the overhead of safely protecting the return addresses of a non-randomized kernel. Finally, the last two columns combine RW+SS with code diversification schemes; RW+SS+CPH with CPH [48] and RW+SS+KASLR with fine-grained KASLR (Section 3.4).

Instrumenting memory writes imposes a maximum overhead of 17.50% (avg. 3.04%) on latency and 4.23% (avg. 1.32%) on bandwidth. Interestingly, instrumenting both memory read and memory write instructions *lowers* the overhead, with a maximum of 7.80% (avg. 2.75) on latency and a maximum of 4.08% (avg. 1.30%) on bandwidth. We attribute that to the check optimization of kR^X (Section 3.4: by "combining" both read and write range checks when eliminating checks, this optimization becomes more effective and therefore reduces the overall overhead. On the other hand, SS imposes up to 20.99% (avg. 8.12%) on latency and up to 4.59% (avg. 2.53%) on bandwidth which when the kSplitStack region is isolated increases to a maximum of 22.51% (avg. 11.34%) on latency and a maximum of 7.13% (avg. 4.23%). Finally, when the kernel code layout is diversified using CPH, the maximum overhead is 20.83% (avg. 11.42%) on latency and 3.99% (avg. 2.72%) on bandwidth, while when fine-grained KASLR is employed, with a maximum of 27.39% (avg. 13.82%) on latency and 5.78% (avg. 3.45%) on bandwidth.

These results indicate that the overall impact of kSplitStack on latency ranges from small to moderate with open()/close(), fork()+exit(), fork()+/bin/sh and UNIX socket I/O suffering the most, while the impact on bandwidth ranges from negligible to small. Additionally, they show that the impact of both randomization schemes is limited with CPH being more efficient since it only affects indirect function calls, in contrast to fine-grained KASLR that blindly diversifies all functions.

4.6.2.2 Macro-Benchmarks

To obtain an understanding of the performance of a kSplitStack-protected system, we employ PTS [162], a suite that offers a plethora of benchmarks and common workload tests of popular real-world applications. From these, we selected a set of tests that stress different types of operations such as serving HTTP requests (Apache), performing transactions on a database (PostgreSQL), building and extracting a kernel (Kbuild and Kextracting), encrypting and signing files (GnuPG and OpenSSL) along with benchmarks that measure the performance of interpreters (PyBench and PHPBench), file system and disk (IOZone

	Benchmark	`W`	`RW`	`SS`	`RW+SS`	`RW+SS+CPH`	`RW+SS+KASLR`
Latency	`syscall()`	~0%	0.43%	3.55%	4.59%	5.32%	5.52%
	`open()/close()`	1.77%	4.71%	9.29%	19.25%	16.32%	22.45%
	`read()/write()`	~0%	0.31%	6.71%	9.83%	9.78%	11.47%
	`select(`10 `fds)`	0.57%	1.15%	8.92%	9.73%	10.38%	13.71%
	`select(`100 TCP `fds)`	8.38%	3.94%	0.48%	5.76%	6.26%	7.21%
	`fstat()`	~0%	0.90%	7.87%	12.38%	11.85%	14.87%
	`mmap()/munmap()`	1.43%	3.48%	5.57%	9.32%	7.68%	9.51%
	`fork()+exit()`	11.47%	7.26%	9.87%	15.26%	15.89%	14.41%
	`fork()+execve()`	~0%	~0%	6.73%	11.25%	18.37%	23.82%
	`fork()+/bin/sh`	17.50%	0.63%	20.99%	22.51%	20.83%	27.39%
	`sigaction()`	~0%	~0%	7.24%	8.08%	7.92%	8.47%
	Signal delivery	0.20%	1.27%	13.19%	14.27%	15.72%	17.15%
	Protection fault	~0%	0.73%	~0%	4.63%	1.45%	2.93%
	Page fault	6.40%	7.80%	8.32%	7.78%	13.07%	15.28%
	Pipe I/O	~0%	2.02%	10.47%	12.75%	10.21%	9.81%
	UNIX socket I/O	6.95%	6.33%	16.82%	20.37%	18.18%	17.79%
	TCP socket I/O	~0%	3.51%	~0%	3.45%	2.08%	9.14%
	UDP socket I/O	~0%	5.06%	10.24%	12.93%	14.30%	17.81%
Bandwidth	Pipe I/O (bandwidth)	0.25%	1.72%	4.08%	3.81%	3.99%	4.75%
	UNIX socket I/O (bandwidth)	0.72%	0.68%	1.22%	2.89%	2.64%	3.06%
	TCP socket I/O (bandwidth)	1.40%	~0%	4.59%	3.12%	3.79%	5.78%
	`mmap()` I/O (bandwidth)	4.23%	4.08%	~0%	4.18%	~0%	~0%
	File I/O (bandwidth)	~0%	~0%	2.74%	7.13%	3.17%	3.66%

Table 4.1: kSplitStack runtime overhead on the LMBench micro-benchmark (% over vanilla Linux; MPX support).

Benchmark	Metric	W	RW	SS	RW+SS	RW+SS+CPH	RW+SS+KASLR
Apache	Req/s	2.81%	1.26%	2.81%	1.62%	1.94%	1.39%
PostgreSQL	Trans/s	1.30%	3.01%	~ 0%	2.52%	2.41%	1.20%
Kbuild	sec	~ 0%	0.81%	1.02%	2.36%	1.69%	2.38%
Kextract	sec	0.39%	~ 0%	0.52%	0.26%	0.56%	0.52%
GnuPG	sec	~ 0%	~ 0%	~ 0%	~ 0%	~ 0%	0.10%
OpenSSL	Sign/s	~ 0%	~ 0%	~ 0%	~ 0%	~ 0%	~ 0%
PyBench	msec	~ 0%	~ 0%	~ 0%	~ 0%	0.05%	0.01%
PHPBench	Score	~ 0%	~ 0%	~ 0%	~ 0%	~ 0%	~ 0%
IOZone	MB/s	1.74%	3.06%	3.97%	11.32%	8.22%	10.13%
DBench	MB/s	3.49%	2.14%	2.17%	1.18%	~ 0%	~ 0%
PostMark	Trans/s	~ 0%	1.82%	11.11%	12.07%	12.60%	16.18%
Average		**0.88%**	**1.10%**	**1.96%**	**2.85%**	**2.50%**	**2.90%**

Table 4.2: kSplitStack runtime overhead on the Phoronix Test Suite (% over vanilla Linux; MPX support).

and `DBench`), and a simulation of an email server which manipulates multiple small files (`PostMark`).

Table 4.2 shows our results. Instrumenting only memory write instructions imposes an average overhead of 0.88% (`W`) which increases to 1.10% when memory read instructions are also instrumented (`RW`). The overhead of employing a shadow stack without isolating the kSplitStack region is 1.96% (`SS`) while when the region is isolated the overhead raises to 2.85% (`RW+SS`). Finally, employing CPH to diversify the code slightly *lowers* the average overhead to 2.50% (`RW+SS+CPH`) while employing KASLR does not add a significant impact with a minor increase to 2.90% (`RW+SS+KASLR`).

Overall, the overhead of kSplitStack on real-world applications is small and comparable to the overhead of kR^X. Note that `PostMark` exhibits significantly larger overhead than the rest of the tests. This is expected: kSplitStack employs the instrumentation of kR^X, which is known to perform worse on `PostMark` than the rest of the tests (Section 3.6.2).

4.6.3 Security Evaluation

In this section we discuss the effectiveness of kSplitStack against any flavor of (kernel) code reuse attacks. Note that for this security discussion we use the complete version of kSplitStack, i.e., RW+SS+CPH.

Direct Code Reuse and Direct JIT Code Reuse kSplitStack relies on the methodology of kR^X to prevent these types of attacks, therefore we employed the same set of experiments as in Section 3.6.3 to assess its effectiveness. In a kSplitStack-protected kernel, direct code reuse exploits would fail due to the fine-grained code diversification (function permutation) employed. In our experiment, the exploit failed and we verified that the addresses of all the gadgets that it employed were relocated (the functions that they belonged were in a different order). Additionally, kSplitStack prevents direct JIT code reuse attacks by rendering the kernel code unreadable, thus our exploit attempt failed at the code leak stage.

Indirect JIT Code Reuse To perform an indirect JIT code reuse attack, an adversary needs to leak code pointers in order to construct the exploit. In the kernel setting, there are three types of code pointers an adversary could employ: return addresses, code pointers emitted during hardware events, and function pointers. kSplitStack stores return addresses in its protected region in a race-free manner, therefore completely mitigating the threat of leaking such code pointers. Additionally, because the return addresses always remain in the protected region which is also not writable, they cannot be corrupted as part of a substitution attack (Section 3.7.1). Similarly, it safely protects code pointers emitted in interrupt context by forcing the hardware through the IST to emit them in the protected region instead of the unprotected kernel stack.

In contrast to kR^X, kSplitStack does not diversify the internal layout of functions thus all gadgets remain in the same offset within the function body. As a result protecting the function start address is of paramount importance. Fortunately, kSplitStack employs CPH which mitigates this issue since any leaked function pointer holds the address of the "trampoline", thus impeding attackers from finding the real location of the function. Note that since the "trampolines" are part of the code section they are not readable, thus an attacker cannot employ her arbitrary memory read to leak their body. Finally, in x86(-64) the target of direct function calls is relative to the address of the call instruction, hence it is not possible to be predicted in order to use the —unaligned— opcodes of the `call` instruction as gadgets [173].

4.7 Discussion

4.7.1 Comparison with CFI

4.7.1.1 Security Analysis

In this section we discuss the difference between kSplitStack (specifically `RW+SS+CPH`) when coupled with a fine-grained CFI solution and a CFI scheme similar to the one proposed in the seminal CFI paper by Abadi et al. [2]. This scheme relies on fine-grained CFI for the protection of the forward edges (function calls) and a shadow stack to protect backwards edges (return addresses). For the purpose of a fair comparison, we will also assume that code pointers emitted by hardware events are also safely stored in the shadow stack, similarly to how kSplitStack protects them. These two schemes seem initially similar in terms of their protection —they both utilize an shadow stack to protect backward edges and code pointers emited during hardware events, while CFI protects forward edges. They do, however, have a significant difference: the code of a kSplitStack-protected kernel is diversified and not readable, whereas CFI imposes no such restriction to the attacker.

Under both schemes it is not possible to reliably redirect the control flow to a gadget in the middle of a function. Additionally, under kSplitStack identifying the location of gadgets inside the body of functions is not a viable option. Attackers therefore have to revert to traditional whole function reuse typically employed in return-to-libc [58] exploits. This methodology relies on corrupting the arguments that are passed during function calls, however in x86-64 the first six arguments are passed through registers [142] an architectural characteristic that severely limits the flexibility of this approach.

	Benchmark	`CFI`	`RW+SS+CPH+CFI`
Latency	`syscall()`	3.03%	4.51%
	`open()/close()`	12.24%	16.51%
	`read()/write()`	7.62%	10.41%
	`select(`10 `fds)`	9.27%	10.90%
	`select(`100 TCP `fds)`	3.85%	8.44%
	`fstat()`	9.12%	11.45%
	`mmap()/munmap()`	8.16%	9.07%
	`fork()+exit()`	14.34%	15.82%
	`fork()+execve()`	17.02%	19.92%
	`fork()+/bin/sh`	20.20%	22.78%
	`sigaction()`	8.69%	7.74%
	Signal delivery	13.26%	15.16%
	Protection fault	0.36%	2.19%
	Page fault	82.36%	66.27%
	Pipe I/O	9.29%	11.59%
	UNIX socket I/O	12.21%	16.80%
	TCP socket I/O	4.93%	4.59%
	UDP socket I/O	12.33%	14.33%
Bandwidth	Pipe I/O (bandwidth)	2.21%	4.01%
	UNIX socket I/O (bandwidth)	2.64%	3.24%
	TCP socket I/O (bandwidth)	3.05%	4.55%
	`mmap()` I/O (bandwidth)	4.16%	2.96%
	File I/O (bandwidth)	25.59%	19.90%

Table 4.3: Comparison of kSplitStack and CFI runtime overhead on the LMBench microbenchmark (% over vanilla Linux; MPX support).

The above force an attacker to attempt overwriting function pointers —since they are not placed in the protected region— as part of a Call Oriented Programming (COP) [23] exploit. While a feasible approach under both schemes, the fine-grained CFI component of both schemes would force the attacker to construct their exploit by overwriting function pointers with targets of the *same signature*, limiting their options. Due to the code diversification and unreadable code under kSplitStack however, the attacker options are further limited; the set of addresses that an attacker can use as targets when overwriting function pointers is limited to the (exposed) address-taken function "trampolines". On kernel v3.19 the address-taken functions amount for only ~26% of the total number of functions that the attacker could utilize under the original CFI scheme.

Benchmark	Metric	CFI	RW+SS+CPH+CFI
Apache	Req/s	2.39%	2.06%
PostgreSQL	Trans/s	~ 0%	1.48%
Kbuild	sec	1.52%	1.83%
Kextract	sec	~ 0%	0.27%
GnuPG	sec	0.16%	0.10%
OpenSSL	Sign/s	~ 0%	~ 0%
PyBench	msec	~ 0%	0.05%
PHPBench	Score	~ 0%	~ 0%
IOZone	MB/s	8.92%	11.10%
DBench	MB/s	~ 0%	~ 0%
PostMark	Trans/s	11.11%	13.27%
Average		**2.19%**	**2.74%**

Table 4.4: Comparison of kSplitStack and CFI runtime overhead on the Phoronix Test Suite (% over vanilla (% over vanilla Linux; MPX support).

4.7.1.2 Performance Analysis

To reduce the number of victim functions that an attacker can use in their exploit kSplitStack needs to instrument memory read instructions and diversify the code layout. Therefore, to further explore this comparison, we implemented both schemes and measured their overhead using the same set of experiments as the one we employed to assess the performance impact of kSplitStack (Section 4.6). Specifically, we coupled kSplitStack with the forward-edge protection of kCFI [146]. Similarly, we implemented a scheme similar to the one of Abadi et al. [2] by coupling the shadow stack of kSplitStack (but only protecting it from corruption attempts, thus instrumenting only memory writes) with kCFI.

Table 4.3 shows the results of LMBench for both schemes. `CFI` represents the overhead of the original CFI scheme, while `RW+SS+CPH+CFI` represents the overhead of kSplitStack when coupled with fine-grained CFI. `CFI` has an average overhead of 13.79% on latency while `RW+SS+CPH+CFI` has an average latency overhead of 14.91%. In both schemes the tests that suffer the most are `fork()+exit()`, `fork()+execve()` and `fork()+/bin/sh` and have a clear outlier on the page fault handler benchmark which amounts as the largest overhead on both (82.36% on `CFI`, 66.27% on `RW+SS+CPH+CFI`). Regarding bandwidth, `CFI` has an average overhead of 7.52% while `RW+SS+CPH+CFI` has an average overhead of 6.93%. In both schemes the test that is affected the most is File I/O (25.59% on `CFI`, 19.90% on `RW+SS+CPH+CFI`).These results show that `RW+SS+CPH+CFI` imposes 1.12% larger overhead on latency but 0.60% lower overhead on bandwidth.

Table 4.4 shows the performance overhead of both schemes when measured on PTS. `CFI` imposes an average overhead of 2.19% with `RW+SS+CPH+CFI` exhibiting a slightly higher average overhead of 2.74%. Both schemes impose their largest overhead on `PostMark`.

Chapter 5

Conclusion

5.1 Summary

In this dissertation, we investigated the hypothesis that the security of modern OSes can be improved by adopting self-protection mechanisms specifically tailored to the kernel setting that minimize the set of code pointer an attacker can tamper with to reliably mount code reuse exploits.

Towards this goal, we presented kR^X: a comprehensive and practical solution against code reuse attacks that target x86 and x86-64 Linux kernels. To prevent simple code reuse exploit attempts it relies on strong code diversification (function and code block permutation). It then instruments every memory read instruction with SFI-inspired range checks which render the loaded code unreadable, thus thwarting direct JIT code reuse attacks. Finally, it protects return addresses through two novel schemes —one based on encryption, the other based on deception— to tackle indirect JIT code reuse exploits. Finally, it takes advantage of new hardware features (i.e., Intel MPX) or architectural characteristics (i.e., segmentation unit on x86) to reduce its performance overhead. Our extensive evaluation demonstrates that kR^X can effectively and efficiently protect kernel software from code reuse attacks with low overhead.

In addition, we presented kSplitStack: a solution that further hardens x86-64 Linux kernel software against indirect JIT code reuse attacks. It solves the race hazards limitations of the return address protection schemes of kR^X through the use of a (specially crafted) shadow stack. Specifically, it enforces race-free return address protection by relocating the stack pointer before the call through a novel code instrumentation scheme. Additionally, it protects hardware emitted code pointers (and other control data) in interrupt context through the use of x86-64 architectural features (i.e., IST). Our experimental evaluation shows that it provides stronger protection against code reuse attacks than kR^X with similarly low overhead.

5.2 Future Directions

The works we presented improve the current stature of OS kernels from a security standpoint however they are far from sufficient to guarantee the security of kernel software. There are many more threats that need to be addressed by the security community. Over the next few paragraphs we discuss some directions that researchers should consider moving towards the next-generation of OSes.

5.2.1 Current and Future Threat Mitigation

Software and more specifically kernel security has been revolving mostly around control flow hijacking attacks, due to either the weak kernel-user segregation [109] or the ability to “repurpose” already existing, legitimate code to perform malicious actions. We believe that the combination of the various defenses in modern systems [43, 82] along with the defenses proposed by the research community [50, 79, 83, 89, 109, 129, 146, 157] and the solutions presented in this dissertation raise the bar significantly. We anticipate that attackers will migrate to different types of attacks which have escaped the attention of the research community, either as stepping stones to bypass deployed defenses or as new attack vectors.

One such threat is data-only attacks [27], which target non-control data. The kernel contains multiple sensitive data structures which can be used both to elevate the privileges of attacker controlled processes (e.g., process credentials), undermine the secrecy of cryptographic keys that reside in the kernel portion of the address space or to disable/bypass defenses (e.g., page tables, control registers). Both attackers [127] and defenders [26, 52, 53] have started exploring this space, however we believe that there is room for more principled solutions. One option is Data-Flow Integrity [24, 176], however current state-of-the-art prototypes exhibit significant overhead. One potential avenue of research would be to place sensitive data structures in isolated regions (similar to the ones of kR^X and kSplitStack) in order to prevent their corruption. As evidenced by our evaluation, this would be sufficiently practical and effective. A potentially less invasive solution could rely on Process-Context Identifiers (PCIDs) [104] to protect sensitive data structures by rendering them unmapped to processes that try to access them through a non-legitimate path.

Another line of attacks that rapidly gain momentum and notoriety is micro-architectural side channel attacks [93, 99, 114, 134, 188]. Such attacks take advantage of vulnerabilities in the implementation of various aspects of processors and leak information without "accessing" them. We believe that since such attacks rely on hardware vulnerabilities, they are not easily mitigated by software (though specific exploits can be prevented [43, 92]), however detecting such attacks could be feasible. We envision a kernel subsystem that would sample various processor counters (e.g., through `perf` [110]) every few milliseconds and pass this information to an anomaly detection Intrusion Detection System (IDS) [75]. While imprecise this mechanism could provide valuable information to detect and pinpoint such attacks.

5.2.2 Security as a Design Principle

More often than not, security mechanisms are added on top of kernel software instead of being "engrained" in the design. Unfortunately, this paradigm favors attackers: vulnerable systems are patched only after attackers expose and exploit their weaknesses. We envision a holistic approach that would place security in the same priority as efficiency and correctness. Microkernels [95], despite their excessive overhead which limits their adoption, are an example of this approach. Another promising avenue is migrating kernel software from memory unsafe languages like C to memory safe ones like Rust [138], as showcased for small kernels [128]. Unfortunately, this would require rewriting an astounding amount of kernel code (Linux kernel v5.2 consists of approximately 18 millions LOC), which makes it a less compelling option. Due to the above, we believe that the community should focus on designing compiler-based defenses which can produce lightweight defenses on legacy code. Making compiler-based security solutions part of the kernel software design and effectively making them part of the fabric, could change the current “cat-and-mouse game” stature of securing kernel software.

Bibliography

[1] Analysis of jailbreakme v3 font exploit. `http://esec-lab.sogeti.com/posts/2011/07/16/analysis-of-the-jailbreakme-v3-font-exploit.html`, July 2011.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proc. of CCS*, pages 340–353, 2005.

[3] ARM Limited. *ARMv8-A Architecture Reference Manual*, March 2015.

[4] 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 *Proc. of CCS*, pages 90–102, 2014.

[5] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can'T Read: Preventing Disclosure Exploits in Executable Code. In *Proc. of ACM CCS*, pages 1342–1353, 2014.

[6] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proc. of USENIX Sec*, pages 433–447, 2014.

[7] Dennis Batchelder, Joe Blackbird, David Felstead, Paul Henry, Jeff Jones, Aneesh Kulkarni, John Lambert, Marc Lauricella, Ken Malcolmson, Matt Miller, Nam Ng, Daryl Pecelj, Tim Rains, Vidya Sekhar, Holly Stewart, Todd Thompson, David Weston, and Terry Zink. How Vulnerabilities are Exploited. *Microsoft Security Intelligence Report*, 16:6–8, December 2013.

[8] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of USENIX OSDI*, pages 423–436, 2010.

[9] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. of USENIX Sec*, pages 255–270, 2005.

[10] Joe Bialek. The Evolution of CFI Attacks and Defenses. `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2018_02_OffensiveCon/The%20Evolution%20of%20CFI%20Attacks%20and%20Defenses.pdf`, February 2018.

[11] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proc. of CCS*, pages 268–279, 2015.

[12] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Maziéres, and Dan Boneh. Hacking blind. In *Proc. of S&P*, pages 227–242, 2014.

[13] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *Proc. of USENIX Summer*, pages 87–98, 1994.

[14] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*, chapter Modules, pages 842–851. O'Reilly Media, 3^{rd} edition, 2005.

[15] Daniel Pierre Bovet. Special sections in Linux binaries. `http://lwn.net/Articles/531148/`, January 2013.

[16] Brad Spengler and Sorbo. Linux `perf_swevent_init` Privilege Escalation. `https://packetstormsecurity.com/files/125674/Linux-perf_swevent_init-Privilege-Escalation.html`.

[17] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proc. of NDSS*, 2016.

[18] David Brash. The ARMv8-A architecture and its ongoing development. `https://community.arm.com/groups/processors/blog/2014/12/02/the-armv8-a-architecture-and-its-ongoing-development`, December 2014.

[19] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to RISC. In *Proc. of CCS*, pages 27–32, 2008.

[20] Adrian Bunk. i386: always enable regparm. `https://goo.gl/uo6taH`, December 2006.

[21] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *Proc. of IEEE S&P*, 2019.

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proc. of USENIX Sec*, pages 161–176, 2015.

[23] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proc. of USENIX Sec*, pages 385–399, 2014.

[24] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proc. of OSDI*, pages 147–160, 2006.

[25] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proc. of CCS*, pages 559–572, 2010.

[26] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proc. of ASIACCS*, pages 167–178, 2017.

[27] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proc. of USENIX Sec.*, pages 177–192, 2005.

[28] Xi Chen, Herbert Bos, and Cristiano Giuffrida. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *Proc. of IEEE EuroS&P*, pages 514–529, 2017.

[29] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proc. of IEEE S&P*, pages 304–319, 2017.

[30] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proc. of ICDCS*, pages 409–417, 2001.

[31] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proc. of CCS*, pages 952–963, 2015.

[32] Kees Cook. x86: make IDT read-only. `https://lkml.org/lkml/2013/4/8/749`, April 2013.

[33] Kees Cook. Kernel Self Protection Project. `http://kernsec.org/wiki/index.php/Kernel_Self_Protection_Project`, January 2016.

[34] F. J. Corbató and V. A. Vyssotsky. Introduction and Overview of the Multics System. In *Proc. of AFIPS*, pages 185–196, 1965.

[35] Jonathan Corbet. Virtual Memory I: the problem. `http://lwn.net/Articles/75174/`, March 2004.

[36] Jonathan Corbet. vmsplice(): the making of a local root exploit. `https://lwn.net/Articles/268783/`, February 2008.

[37] Jonathan Corbet. An updated guide to `debugfs`. `http://lwn.net/Articles/334546/`, May 2009.

[38] Jonathan Corbet. A JIT for packet filters. `https://lwn.net/Articles/437981/`, April 2011.

[39] Jonathan Corbet. Supervisor mode access prevention. `http://lwn.net/Articles/517475/`, October 2012.

[40] Jonathan Corbet. https://lwn.net/Articles/599755/. `https://lwn.net/Articles/599755/`, May 2014.

[41] Jonathan Corbet. Supporting Intel MPX in Linux. `https://lwn.net/Articles/582712/`, January 2014.

[42] Jonathan Corbet. A rough patch for live patching. `https://lwn.net/Articles/634649/`, February 2015.

[43] Jonathan Corbet. The current state of kernel page-table isolation. `http://lwn.net/Articles/741878/`, December 2017.

[44] Jonathan Corbet, Greg Kroah-Hartman, and Amanda McPherson. Linux Kernel Development. Technical report, Linux Foundation, October 2017.

[45] Marc L. Corliss, 05 Christopher Lewis, and Amir Roth. Using dise to protect return addresses from attack. *ACM SIGARCH Computer Architecture News*, 33(1):5:1–5:28, March 2005.

[46] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of USENIX Sec.*, pages 63–78, 1998.

[47] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. Booby Trapping Software. In *Proc. of NSPW*, pages 95–106, 2013.

[48] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proc. of IEEE S&P*, pages 763–780, 2015.

[49] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In *Proc. of ACM CCS*, pages 243–255, 2015.

[50] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proc. of IEEE S&P*, pages 292–307, 2014.

[51] Thurston H.Y Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proc. of ASIACCS*, pages 555–566, 2015.

[52] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proc. of ASPLOS*, pages 191–206, 2015.

[53] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. PT-Rand: Practical Mitigation of Data-only Attacks against Page Tables. In *Proc. of NDSS*, 2017.

[54] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proc. of NDSS*, 2015.

[55] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proc. of USENIX Sec*, pages 401–416, 2014.

[56] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proc. of ASIACCS*, pages 40–51, 2011.

[57] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *Proc. of ASIACCS*, pages 299–310, 2013.

[58] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, August 1997.

[59] Jake Edge. Kernel address space layout randomization. `http://lwn.net/Articles/569635/`, October 2013.

[60] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Proc. of ACM SOSP*, pages 133–150, 2013.

[61] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proc. of CCS*, pages 901–913, 2015.

[62] Evans, Isaac and Fingeret, Sam and Gonzalez, Julian and Otgonbaatar, Ulziibayar and Tang, Tiffany and Shrobe, Howard E. and Sidiroglou, Stelios and Rinard, Martin C. and Okhravi, Hamed. Missing the point(er): On the effectiveness of code pointer integrity. In *Proc. of IEEE S&P*, pages 781–796, 2015.

[63] Exploit Database. EBD-131, December 2003.

[64] Exploit Database. EBD-14814, August 2010.

[65] Exploit Database. EBD-15150, September 2010.

[66] Exploit Database. EBD-15285, October 2010.

[67] Exploit Database. EBD-15916, January 2011.

[68] Exploit Database. EBD-17787, September 2011.

[69] Exploit Database. EBD-20201, August 2012.

[70] Exploit Database. EBD-24555, February 2013.

[71] Exploit Database. EBD-31346, February 2014.

[72] Exploit Database. EBD-33516, May 2014.

[73] FireFart. 'Dirty COW PTRACE_POKEDATA' Race Condition Privilege Escalation. https://www.exploit-db.com/exploits/40839, November 2016.

[74] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *Proc. of USENIX Sec*, pages 83–97, 2018.

[75] Pedro García-Teodoro, Jesús Díaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1–2):18–28, February–March 2009.

[76] Thomas Garnier. x86: Make the GDT remapping read-only on 64-bit. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=45fc8757d1d2128e342b4e7ef39adedf7752faac, March 2017.

[77] GCC online documentation. Intel 386 and AMD x86-64 Options. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/i386-and-x86_002d64-Options.html.

[78] Xinyang Ge, Mathias Payer, and Trent Jaeger. An Evil Copy: How the Loader Betrays You. In *Proc. of NDSS*, 2017.

[79] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proc. of IEEE EuroS&P*, 2016.

[80] Jason Geffner. VENOM: Virtualized Environment Neglected Operations Manipulation. http://venom.crowdstrike.com, May 2015.

[81] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *Proc. of CCS*, pages 133–144, 2012.

[82] Varghese George, Tom Piazza, and Hong Jiang. Technology Insight: Intel® Next Generation Microarchitecture Codename Ivy Bridge. http://www.intel.com/idf/library/pdf/sf_2011/SF11_SPCS005_101F.pdf, September 2011.

[83] Jason Gionta, William Enck, and Per Larsen. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *Proc. of IEEE CNS*, 2016.

[84] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proc. of ACM CODASPY*, pages 325–336, 2015.

[85] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proc. of USENIX Sec*, pages 475–490, 2012.

[86] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.

[87] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining Information Hiding (And What to do About it). In *Proc. of USENIX Sec*, pages 105–119, 2016.

[88] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the Line: Practical Cache Attacks on the MMU. In *Proc. of NDSS*, 2017.

[89] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. IskiOS: Lightweight Defense Against Kernel-Level Code-Reuse Attacks. `http://arxiv.org/abs/1903.04654`, March 2019.

[90] grsecurity. Active kernel exploit response. `https://xorl.wordpress.com/2011/04/27/grkernsec_kern_lockout-active-kernel-exploit-response/`, April 2011.

[91] Daniel Gruss. *Software-based Microarchitectural Attacks.* PhD thesis, Graz University of Technology, 2017.

[92] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, and Stefan Maurice, Clémentine Mangard. KASLR is Dead: Long Live KASLR. In *Proc. of ESSoS*, pages 161–176, 2017.

[93] Daniel Gruss, Clémentine Maurice, Andreas Fogh, Moritz Lipp, and Stefan Mangard. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proc. of ACM CCS*, pages 368–379, 2016.

[94] Dave Hansen. [RFC] x86: Memory protection keys. `https://lwn.net/Articles/643617/`, May 2015.

[95] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A Highly Reliable, Self-Repairing Operating System. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, July 2006.

[96] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J.W. Davidson. ILR: Where'd My Gadgets Go? In *Proc. of IEEE S&P*, pages 571–585, 2012.

[97] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proc. of CCS*, pages 1470–1486, 2018.

[98] Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec*, pages 384–398, 2009.

[99] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Proc. of IEEE S&P*, pages 191–205, 2013.

[100] Ralph Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *Proc. of IEEE S&P*, pages 191–205, 2013.

[101] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, April 2015.

[102] Intel Corporation. *Intel® Memory Protection Extensions Enabling Guide*, January 2016.

[103] Intel Corporation. Control-flow Enforcement Technology Preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, June 2017.

[104] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, May 2019.

[105] International Business Machines (IBM). *Power ISA^{TM} Version 3.0 B*, March 2017.

[106] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *Proc. of ACM CCS*, pages 380–392, 2016.

[107] Paul A. Karger and Andrew J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. of IEEE S&P*, pages 2–12, 1984.

[108] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. of USENIX Sec*, pages 957–972, 2014.

[109] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proc. of USENIX Sec*, pages 459–474, 2012.

[110] Kernel.org. Perf_events tutorial. `https://perf.wiki.kernel.org`, September 2015.

[111] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proc. of ACSAC*, pages 339–348, 2006.

[112] Thomas J. Killian. Processes as Files. In *Proc. of USENIX Summer*, pages 203–207, 1984.

[113] Andi Kleen. Memory Layout on amd64 Linux. `https://www.kernel.org/doc/Documentation/x86/x86_64/mm.txt`, July 2004.

[114] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *To appear in Proc. of IEEE S&P*, May 2019.

[115] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. Architecture Support for Single Address Space Operating Systems. In *Proc. of ASPLOS*, pages 175–186, 1992.

[116] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proc. of EuroSys*, pages 437–452, 2017.

[117] Andrey Konovalov. DCCP Double-Free Privilege Escalation. `https://www.exploit-db.com/exploits/41458`, February 2016.

[118] Andrey Konovalov. Packet Socket Privilege Escalation. `https://www.exploit-db.com/exploits/41994`, May 2017.

[119] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *Proc. of IEEE S&P*, pages 472–488, 2018.

[120] Tim Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-University, 2009.

[121] Mathias Krause. CVE Requests (maybe): Linux kernel: various info leaks, some `NULL` ptr derefs. `http://www.openwall.com/lists/oss-security/2013/03/05/13`, March 2013.

[122] Greg Kroah-Hartman. `udev` – A Userspace Implementation of devfs. In *Proc. of OLS*, pages 263–271, 2003.

[123] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proc. of NDSS*, 2013.

[124] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proc. of USENIX OSDI*, pages 147–163, 2014.

[125] Mike Larkin. Kernel W^X Improvements In OpenBSD. In *Hackfest*, 2015.

[126] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated Software Diversity. In *Proc. of IEEE S&P*, pages 276–291, 2014.

[127] JungSeung Lee, HyoungMin Ham, InHwan Kim, and JooSeok Song. POSTER: Page Table Manipulation Attack. In *Proc. of ACM CCS*, pages 1644–1646, 2015.

[128] Amit Levy, Bradford Campbell, Branded Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proc of SOSP*, pages 234–251, 2017.

[129] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-CFI: Fine-grained Control-Flow Integrity for Operating System Kernels. *IEEE Trans. Inf. Forensics Security*, 13(6):1535–1550, June 2018.

[130] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, and Sina Bahram. Defeating Return-Oriented Rootkits With "*Return-less*" Kernels. In *Proc. of EuroSys*, pages 195–208, 2010.

[131] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *Proc. of ATC*, pages 1–13, 2017.

[132] Siarhei Liakh. NX protection for kernel data. `http://lwn.net/Articles/342266/`, July 2009.

[133] Linux Cross Reference. Linux kernel release 3.19. `http://lxr.free-electrons.com/source/arch/x86/kernel/cpu/perf_event_intel_uncore_snb.c?v=3.19#L565`.

[134] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proc. of USENIX Sec*, pages 973–990, August 2018.

[135] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. 2016.

[136] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proc. of CCS*, pages 280–291, 2015.

[137] 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 *Proc. of ACM PLDI*, pages 190–200, 2005.

[138] Nicholas D. Matsakis and Felix S. Klockm II. The rust language. In *Proc. of ACM HILT*, pages 103–104, 2014.

[139] Matthew Gillespie. Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d. `https://software.intel.com/en-us/articles/best-practices-for-paravirtualization-enhancements-from-intel-virtualization-technology-ept-and-vt-d`, January 2015.

[140] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC Architecture. In *Proc. of USENIX Sec*, pages 209–224, 2006.

[141] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of USENIX ATC*, pages 279–294, 1996.

[142] Michael Matz and Jan Hubička and Andreas Jaeger and Mark Mitchell. System V Application Binary Interface. `http://www.x86-64.org/documentation/abi.pdf`, October 2013.

[143] Microsoft. Data Execution Prevention (DEP). `http://goo.gl/38j1fT`, 2006.

[144] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. Microstache: A lightweight execution context for in-process safe region isolation. In *Proc. of RAID*, pages 359–379, 2018.

[145] Ingo Molnar. 4G/4G split on x86, 64 GB RAM (and more) support. `http://lwn.net/Articles/39283/`, July 2003.

[146] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios P. Kemerlis. Drop the rop: Finegrained control-flow integrity for the linux kernel. 2017.

[147] National Vulnerability Database. Kernel Vulnerabilities. `https://goo.gl/K1hYTh`, July 2019.

[148] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking Holes in Information Hiding. In *Proc. of USENIX Sec*, pages 121–138, 2016.

[149] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[150] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proc. of IEEE S&P*, pages 601–615, 2012.

[151] PaX. Homepage of The PaX Team. `http://pax.grsecurity.net`.

[152] PaX Team. ASLR. `https://pax.grsecurity.net/docs/aslr.txt`, July 2001.

[153] PaX Team. NOEXEC. `https://pax.grsecurity.net/docs/noexec.txt`, May 2003.

[154] PaX Team. UDEREF/i386. `http://grsecurity.net/~spender/uderef.txt`, April 2007.

[155] PaX Team. UDEREF/amd64. `http://grsecurity.net/pipermail/grsecurity/2010-April/001024.html`, April 2010.

[156] PaX Team. Better kernels with GCC plugins. `http://lwn.net/Articles/461811/`, October 2011.

[157] PaX Team. RAP: RIP ROP. In *Hackers 2 Hackers Conference (H2HC)*, 2015.

[158] Nick L. Petroni, Jr. and Michael Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. of CCS*, pages 103–115, 2007.

[159] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. DynaGuard: Armoring Canary-based Protections against Brute-force Attacks. In *Proc. of ACSAC*, pages 351–360, 2015.

[160] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. Breaking and Fixing Destructive Code Read Defenses. In *Proc. of ACSAC*, pages 55–67, 2017.

[161] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proc. of USENIX Sec*, pages 231–242, 2003.

[162] PTS. Phoronix Test Suite. `http://www.phoronix-test-suite.com`.

[163] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proc. of RAID*, pages 1–20, 2008.

[164] Dan Rosenberg. `kptr_restrict` for hiding kernel pointers. `http://lwn.net/Articles/420403/`, December 2010.

[165] Chris Salls. Linux Kernel 4.13 (Ubuntu 17.10) - '`waitid()`' SMEP/SMAP/Chrome Sandbox Privilege Escalation. `https://www.exploit-db.com/exploits/43127/`, November 2017.

[166] Chris Salls. 'waitid()' SMEP/SMAP/Chrome Sandbox Privilege Escalation. `https://www.exploit-db.com/exploits/43127`, November 2017.

[167] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of IEEE*, 63(9):1278–1308, September 1975.

[168] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proc. of ACSAC*, pages 448–459, 2016.

[169] Michael D. Schroeder and Jerome H. Saltzer. A Hardware Architecture for Implementing Protection Rings. *Commun. ACM*, 15(3):157–170, March 1972.

[170] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of IEEE S&P*, pages 745–762, 2015.

[171] SecurityFocus. Linux Kernel 'perf_counter_open()' Local Buffer Overflow Vulnerability, September 2009.

[172] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proc. of USENIX Sec*, pages 1–11, 2010.

[173] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of CCS*, pages 552–61, 2007.

[174] 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 *Proc. of IEEE S&P*, pages 574–588, 2013.

[175] Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proc. of IEEE S&P*, pages 954–968, 2016.

[176] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. 2016.

[177] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-flow Isolation. In *Proc. of IEEE S&P*, pages 1–17, 2016.

[178] Brad Spengler. Recent ARM security improvements. `https://forums.grsecurity.net/viewtopic.php?f=7&t=3292`, February 2013.

[179] Brad Spengler. Enlightenment Linux Kernel Exploitation Framework. `https://grsecurity.net/~spender/exploits/enlightenment.tgz`, December 2014.

[180] Paul Starzetz. Linux kernel do_mremap VMA limit local privilege escalation vulnerability. `http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt`, March 2004.

[181] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *Proc. of CCS*, pages 256–267, 2015.

[182] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of ASPLOS*, pages 168–177, 2000.

[183] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX Sec*, pages 941–955, 2014.

[184] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *To appear in Proc. of USENIX Sec*, 2019.

[185] Arjan van de Ven. Debug option to write-protect rodata: the write protect logic and config option. `http://lkml.indiana.edu/hypermail/linux/kernel/0511.0/2165.html`, November 2005.

[186] Arjan van de Ven. Add `-fstack-protector` support to the kernel. `http://lwn.net/Articles/193307/`, July 2006.

[187] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proc. of ACM CCS*, pages 1675–1689, 2017.

[188] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue inflight data load. In *Proc. of IEEE S&P*, 2019.

[189] Vendicator. Stack shield. `http://www.angelfire.com/sk/stackshield/info.html`, January 2000.

[190] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-based Fault Isolation. In *Proc. of SOSP*, pages 203–216, 1993.

[191] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. SecPod: a Framework for Virtualization-based Security Systems. In *Proc. of USENIX ATC*, pages 347–360, 2015.

[192] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 380–395, 2010.

[193] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of CCS*, pages 157–168, 2012.

[194] Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proc. of ACM ASIACCS*.

[195] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proc. of USENIX OSDI*, pages 367–382, 2016.

[196] Rafal Wojtczuk. Exploiting "BadIRET" vulnerability (CVE-2014-9322, Linux kernel privilege escalation). `https://goo.gl/bSEhBI`, February 2015.

[197] Henry Wong. Microbenchmarking return address branch prediction. `http://blog.stuffedcow.net/2018/04/ras-microbenchmarks`, April 2018.

[198] Wen Xu and Yubin Fu. Own Your Android! Yet Another Universal Root. In *Proc. of USENIX WOOT*, 2015.

[199] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. of IEEE S&P*, pages 79–93, 2009.

[200] Fenghua Yu. Enable/Disable Supervisor Mode Execution Protection. `http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=de5397ad5b9ad22e2401c4dacdf1bb3b19c05679`, May 2011.

[201] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. HyperCheck: A Hardware-assisted Integrity Monitor. *IEEE Transactions on Dependable and Secure Computing*, 11(4):332–344, July/August 2014.

[202] Mingwei Zhang, Michalis Polychronakis, and R. Sekar. Protecting COTS Binaries from Disclosure-guided Code Reuse Attacks. In *Proc. of ACSAC*, pages 128–140, 2017.

[203] CVE-2010-2963, October 2010.

[204] CVE-2010-3437, September 2010.

[205] CVE-2011-1021, February 2011.

[206] CVE-2013-2094, February 2013.

[207] CVE-2013-6282, October 2013.

[208] CVE-2015-3036, April 2015.

[209] CVE-2015-3290, April 2015.

[210] CVE-2016-10088, December 2016.

[211] CVE-2016-9576, December 2016.

[212] CVE-2017-11473, July 2017.

[213] CVE-2017-11817, July 2017.

[214] CVE-2017-11831, July 2017.

[215] CVE-2017-12762, August 2017.

[216] CVE-2017-15102, October 2017.

[217] CVE-2017-15649, October 2017.

[218] CVE-2017-2636, December 2017.

[219] CVE-2017-5550, January 2017.

[220] CVE-2017-5576, March 2017.

[221] CVE-2017-6001, February 2017.

[222] CVE-2017-6874, March 2017.

[223] CVE-2017-6987, March 2017.

[224] CVE-2017-7022, March 2017.

[225] CVE-2017-7029, March 2017.

[226] CVE-2017-7069, March 2017.

[227] CVE-2017-7114, March 2017.

[228] CVE-2017-7277, March 2017.

[229] CVE-2017-7374, March 2017.

[230] CVE-2017-7477, April 2017.

[231] CVE-2017-7616, April 2017.

[232] CVE-2017-7895, April 2017.

[233] CVE-2017-8465, May 2017.

[234] CVE-2017-8484, May 2017.

[235] CVE-2017-8494, May 2017.

[236] CVE-2017-8719, May 2017.

[237] CVE-2017-8924, May 2017.

[238] CVE-2017-9150, May 2017.

Appendix A

Discovered Kernel Bugs

During the development of kR^X-KAS, we discovered two kernel bugs. The first one, which is security critical, results in memory being accidentally marked as *executable*. In the x86 architecture, the MMU utilizes a multi-level page table hierarchy for mapping virtual to physical addresses. When the Physical Address Extension (PAE) [101] mode is enabled, which is the default nowadays as non-executable protection is only available under PAE mode, each page table entry is 64-bit wide, and except from addressing information also holds flags that define properties of the mapped page(s) (e.g., PRESENT, ACCESSED). Often, multiple adjacent pages sharing the same flags are coalesced to larger memory areas (e.g., 512 4KB pages can be combined to form a single 2MB page) to reduce TLB pollution [60].

This aggregation takes place in the whole kernel address space, including the dynamic, on-demand memory regions, such as the vmalloc arena, which may enforce different protections to (sub)parts of their allocated chunks. Linux uses the pgprot_large_2_4k() and pgprot_4k_2_large() routines for copying the flags from 2MB to 4KB pages, and vice versa, using a local variable (val) to construct an equivalent flags mask. Unfortunately, val is declared as unsigned long, which is 64-bit wide in x86-64 systems, but only 32-bit wide in x86 systems. As a result, the "eXecute-Disable" (XD) bit (most significant bit on each page table entry) is always cleared in the resulting flags mask, marking the respective pages as executable. Since many of these pages may also be writable, this is a critical vulnerability (W^X violation).

The second bug we discovered is related to module loading. Specifically, before a module is loaded, the module loader-linker first checks whether the image of the module fits within the `modules` region. This check is performed inside the `module_alloc()` routine, using the `MODULES_LEN` macro, which holds the total size of the `modules` region. However, in 32-bit (x86) kernels this macro was mistakenly assigned its complementary value, and hence the (sanity) check will *never* fail. Fortunately, this bug does not constitute a vulnerability because a subsequent call to `__vmalloc_node_range()` (which performs the actual memory allocation for each module) will fail if the remaining space in the `modules` region is less than the requested memory (i.e., the size of the module's image).

Appendix B

kSplitStack Performance on Legacy Hardware

In this section we discuss the performance of kSplitStack when MPX support is not available. We employ the same testbed as in Section 4.6 but instead of using the MPX-based range checks, we employ the SFI (O3) range checks (Section 3.4).

Table B.1 summarizes our LMBench results. Instrumenting only memory writes (`W`) incurs an overhead of up to 14.92% (avg. 4.68%) on latency and 4.97% (avg. 1.47%) on bandwidth. Contrary to the results when MPX support is enabled, instrumenting both memory reads and writes (`RW`) *increases* the overhead up to 25.94% (avg. 8.77%) on latency and up to 9.77% (avg. 3.51%) on bandwidth. This implies that the overhead of the SFI range checks is large enough to not be affected by the aggressive check elimination optimization. When this isolation instrumentation is coupled with the kSplitStack return address protection instrumentation (`RW+SS`), the overhead raises to up to 39.08% (avg. 17.65%) on latency and up to 8.13% (avg. 4.77%) on bandwidth, which is approximately the sum of its two components (`RW` and `SS`). Finally, when coupled with the CPH code diversification scheme (`RW+SS+CPH`) the overhead raises slightly to a maximum of 39.25% (avg. 18.09%) on latency and a maximum of 9.01% (avg. 4.61%) on bandwidth, while when coupled with fine-grained KASLR (`RW+SS+KASLR`) the overhead raises more to a maximum of 45.75% (avg. 20.97%) on latency and to a maximum of 10.79% (avg. 5.37%) on bandwidth.

	Benchmark	`W`	`RW`	`RW+SS`	`RW+SS+CPH`	`RW+SS+KASLR`
Latency	`syscall()`	~0%	~0%	4.44%	3.58%	3.92%
	`open()/close()`	11.12%	25.94%	39.08%	39.25%	45.75%
	`read()/write()`	~0%	19.01%	29.76%	29.24%	31.55%
	`select(`10 `fds)`	~0%	10.92%	20.38%	20.66%	22.21%
	`select(`100 TCP `fds)`	3.38%	5.17%	1.80%	4.98%	8.88%
	`fstat()`	~0%	~0%	9.88%	10.02%	13.06%
	`mmap()/munmap()`	1.66%	1.08%	9.45%	9.27%	10.53%
	`fork()+exit()`	6.46%	8.62%	12.90%	18.95%	23.11%
	`fork()+execve()`	12.53%	2.62%	26.64%	23.42%	32.27%
	`fork()+/bin/sh`	14.92%	17.83%	28.65%	26.49%	32.64%
	`sigaction()`	0.10%	0.55%	7.73%	7.34%	8.34%
	Signal delivery	5.16%	10.11%	22.27%	21.66%	23.33%
	Protection fault	~0%	1.59%	3.61%	6.37%	6.67%
	Page fault	8.92%	6.33%	17.41%	17.27%	19.28%
	Pipe I/O	3.89%	13.71%	15.69%	16.49%	16.81%
	UNIX socket I/O	4.20%	13.60%	25.01%	27.17%	28.59%
	TCP socket I/O	7.19%	11.91%	14.49%	16.47%	21.86%
	UDP socket I/O	4.79%	8.89%	28.53%	26.96%	28.75%
Bandwidth	Pipe I/O (bandwidth)	2.34%	4.67%	7.95%	6.63%	7.51%
	UNIX socket I/O (bandwidth)	~0%	2.32%	4.32%	4.21%	4.29%
	TCP socket I/O (bandwidth)	4.97%	9.77%	8.13%	9.01%	10.79%
	`mmap()` I/O (bandwidth)	0.06%	0.09%	0.03%	~0%	0.09%
	File I/O (bandwidth)	~0%	0.69%	3.45%	3.20%	4.19%

Table B.1: kSplitStack runtime overhead on the LMBench micro-benchmark (% over vanilla Linux; no MPX support).

Benchmark	Metric	W	RW	RW+SS	RW+SS+CPH	RW+SS+KASLR
Apache	Req/s	3.90%	3.34%	~ 0%	0.98%	1.84%
PostgreSQL	Trans/s	5.25%	~ 0%	2.66%	1.04%	~ 0%
Kbuild	sec	0.29%	0.77%	1.61%	1.94%	2.69%
Kextract	sec	~ 0%	0.26%	0.26%	0.89%	0.80%
GnuPG	sec	~ 0%	~ 0%	~ 0%	0.11%	0.10%
OpenSSL	Sign/s	~ 0%	~ 0%	~ 0%	0.01%	~ 0%
PyBench	msec	~ 0%	~ 0%	~ 0%	~ 0%	0.07%
PHPBench	Score	~ 0%	~ 0%	0.01%	~ 0%	~ 0%
IOZone	MB/s	~ 0%	4.22%	14.27%	12.50%	12.67%
DBench	MB/s	2.63%	~ 0%	~ 0%	~ 0%	~ 0%
PostMark	Trans/s	6.96%	13.98%	23.07%	22.81%	25.41%
Average		**1.73%**	**2.05%**	**3.81%**	**3.66%**	**3.96%**

Table B.2: kSplitStack runtime overhead on the Phoronix Test Suite (% over vanilla Linux; no MPX support.)

Table B.2 summarizes our PTS results. Instrumenting only memory writes (W) incurs an average overhead of 1.73%, while when instrumenting both memory reads and writes incurs an average overhead of 2.05%. When coupled with the return address protection scheme of kSplitStack (RW+SS) imposes an average overhead of 3.81%. Mirroring the results when MPX support is available, RW+SS+CPH *lowers* the average overhead to 3.66% and RW+SS+KASLR slightly raises it to 3.96%.