Kernel Protection Against Just-In-Time Code Reuse

MARIOS POMONIS and THEOFILOS PETSIOS, Columbia University
ANGELOS D. KEROMYTIS, Georgia Institute of Technology
MICHALIS POLYCHRONAKIS, Stony Brook University
VASILEIOS P. KEMERLIS, Brown University

The abundance of memory corruption and disclosure vulnerabilities in kernel code necessitates the deployment of hardening techniques to prevent privilege escalation attacks. As stricter memory isolation mechanisms between the kernel and user space become commonplace, attackers increasingly rely on code reuse techniques to exploit kernel vulnerabilities. Contrary to similar attacks in more restrictive settings, as in web browsers, in kernel exploitation, non-privileged local adversaries have great flexibility in abusing memory disclosure vulnerabilities to dynamically discover, or infer, the location of code snippets in order to construct code-reuse payloads. Recent studies have shown that the coupling of code diversification with the enforcement of a "read XOR execute" (R^X) memory safety policy is an effective defense against the exploitation of userland software, but so far this approach has not been applied for the protection of the kernel itself.

In this article, we fill this gap by presenting kR^X: a kernel-hardening scheme based on execute-only memory and code diversification. We study a previously unexplored point in the design space, where a hypervisor or a super-privileged component is not required. Implemented mostly as a set of GCC plugins, kR^X is readily applicable to x86 Linux kernels (both 32b and 64b) and can benefit from hardware support (segmentation on x86, MPX on x86-64) to optimize performance. In full protection mode, kR^X incurs a low runtime overhead of 4.04%, which drops to 2.32% when MPX is available, and 1.32% when memory segmentation is in use.

CCS Concepts: • **Security and privacy → Operating systems security**; *Software security engineering*;

Additional Key Words and Phrases: Execute-only memory, code diversification

ACM Reference format:
Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2019. Kernel Protection Against Just-In-Time Code Reuse. *ACM Trans. Priv. Secur.* 22, 1, Article 5 (January 2019), 28 pages.
https://doi.org/10.1145/3277592

5

M. Pomonis is also with Brown University.
kR^X is available at http://nsl.cs.columbia.edu/projects/krx.
This work was supported in part by the National Science Foundation (NSF) through awards CNS-13-18415 and CNS-17-49895, the Office of Naval Research (ONR) through awards N00014-15-1-2378, N00014-17-1-2788, and N00014-17-1-2891, and the Defense Advanced Research Projects Agency (DARPA) through awards D18AP00045 and HR001118C0017, with additional support by Qualcomm. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, NSF, ONR, DARPA, or Qualcomm.
Authors' addresses: M. Pomonis and T. Petsios, Columbia University; emails: {mpomonis, theofilos}@cs.columbia.edu; A. D. Keromytis, Georgia Institute of Technology; email: angelos@gatech.edu; M. Polychronakis, Stony Brook University; email: mikepo@cs.stonybrook.edu; V. P. Kemerlis, Brown University; email: vpk@cs.brown.edu.

2471-2566/2019/01-ART5 $15.00
https://doi.org/10.1145/3277592

1 INTRODUCTION

The deployment of standard kernel-hardening schemes, such as address space layout randomization (KASLR) [50] and non-executable memory [91], has prompted a shift from legacy code injection (in kernel space) to *return-to-user* (ret2usr) attacks [80]. Due to the weak separation between kernel and user space—as a result of the kernel being mapped inside the (upper part of the) address space of every user process for performance reasons—in ret2usr attacks, the kernel control/dataflow is hijacked and redirected to code/data residing in user space, effectively bypassing KASLR and (kernel-space) W^X. Fortunately, however, recent software [38, 69, 80, 116] and hardware [35, 77] kernel protection mechanisms mitigate ret2usr threats by enforcing a more stringent address space separation. Alas, mirroring the co-evolution of attacks and defenses in user space, kernel exploits have started to rely on code-reuse techniques, such as return-oriented programming (ROP) [134, 144]; old ret2usr exploits [14] are converted to use ROP payloads instead of shellcode [18], while modern jailbreak and privilege escalation exploits rely solely on code reuse [2, 129, 152].

At the same time, the security community started developing protections against code-reuse attacks: control-flow integrity (CFI) [7] and code-diversification [44, 72, 87, 112, 149] schemes have been applied both in the user and kernel settings [42, 58, 64, 94]. Unfortunately, these solutions are not bulletproof; both coarse-grained [113, 156, 159] and fine-grained [49, 110, 111, 118, 122, 141] CFI schemes can be bypassed by confining the hijacked control flow to valid execution paths [21, 47, 51, 65, 66], while code diversification can be circumvented by leveraging *memory disclosure* vulnerabilities [135].

Having the ability to disclose the memory contents of a process, exploit code can dynamically pinpoint the exact location of ROP gadgets at runtime and assemble them on the fly into a functional ROP payload. This kind of "just-in-time" ROP (JIT-ROP) [135] is particularly effective against applications with integrated scripting support, such as web browsers. Specifically, by embedding malicious script code into a web page, an attacker can combine a memory disclosure with a corruption bug to enumerate the address space of the browser for gadgets and divert its execution into dynamically constructed ROP code. However, in kernel exploitation, a local (unprivileged) adversary, armed with an arbitrary (kernel-level) memory disclosure vulnerability [84, 99] has increased flexibility in mounting a JIT-ROP attack on a diversified kernel [64], as *any* user program may attack the OS. Therefore, kernel JIT-ROP attacks are not only easier to mount but are also facilitated by the abundance of memory disclosure vulnerabilities in kernel code [88, 100, 102, 108].

As a response to JIT-ROP attacks in user applications, execute-only memory prevents the (on-the-fly) discovery of gadgets by blocking read access to executable pages [27]. Nevertheless, given that widely used CPU architectures such as the x86 do not provide native support for enforcing execute-only permissions, such memory protection(s) can be achieved by relying on page table manipulation [9], TLB desynchronization [63], hardware virtualization [40, 62], or techniques inspired by software-fault isolation (SFI) [86]. A common characteristic of these schemes (with the exception of LR^2 [19]) is that they rely on a more privileged domain (e.g., the OS kernel [9, 63] or a hypervisor [40, 62]) to protect a less privileged domain—in fact, most of the existing approaches are exclusively tailored for user processes. As JIT-ROP-like attacks are expected to become prevalent in the kernel setting, the need for an effective kernel defense against them becomes imperative.

Retrofitting existing hypervisor-based approaches with kernel protection(s) can be an option [62], but this approach comes with several drawbacks. First, when implemented as a special-purpose hypervisor, such a hierarchically privileged scheme may clash with existing hypervisor deployments, requiring nesting two or more hypervisors, and thereby resulting in high runtime overheads [11]. Second, when implemented as part of an existing hypervisor [62], it would increase

not only the virtualization overhead but also the trusted computing base. Finally, in architectures that lack hardware support, efficient virtualization might not be an option at all. On the other hand, the address space layouts imposed by SFI-based schemes, such as NaCl [154] and LR^2 [19], along with other design decisions that we discuss in Section 5.1.1, are non-applicable in the kernel setting, while Intel's upcoming memory Protection Keys for Userspace (PKU) hardware feature, which can be used to enforce execute-only memory in x86 CPUs, is available to userland software only [71].

In this article, 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 SFI-like checks 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 [81, 149] to thwart the direct use of preselected gadgets; and (ii) return-address protection using either an XOR-based encryption scheme [19, 117, 151] or decoy return addresses to thwart gadget inference through saved return addresses on the kernel stacks [28]. *Practical applicability* to existing systems is ensured given that kR^X (i) does not rely on more privileged entities (e.g., a hypervisor [40, 62]) than the kernel itself; (ii) is readily applicable on x86 systems (both 32b and 64b) and can leverage support for memory segmentation or protection (i.e., Intel's MPX [76]) 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.

2 BACKGROUND

Kernel Exploitation. The execution model imposed by the *shared* virtual memory layout between the kernel and user space makes kernel exploitation a fundamentally different craft from the exploitation of userland software. The shared address space provides a vantage point to local attackers, as it enables them to control part of the kernel-accessible memory (i.e., the user space part) [80]. In particular, they can execute code with kernel rights by hijacking a kernel control path and redirecting it to user space, effectively invalidating kernel-space ASLR [50] and W^X [91]. Attacks of this kind, known as *return-to-user (ret2usr)*, can be traced back to the early 1970s [125]. Recently, however, ret2usr has been promoted to the de facto kernel exploitation technique [119].

During a ret2usr attack, kernel data is overwritten with user-space addresses by (ab)using memory corruption vulnerabilities in kernel code. Attackers aim for *control data*, such as return addresses [132], function pointers [138], and dispatch tables [52], because these facilitate code execution. Nevertheless, pointers to *critical data structures* stored in the kernel data section or heap (i.e., non-control data [145]) are also targets, as they enable attackers to tamper with the data contained in certain objects by mapping fake copies in user space [54]. The targeted data structures typically contain data that affect the control flow (e.g., code pointers) to diverge execution to arbitrary locations. The net result of all ret2usr attacks is that the control/dataflow of the kernel is hijacked and redirected to user space code/data [80].

Code Reuse Prevention. Code reuse exploits rely on code fragments (gadgets) located at *predetermined* memory addresses [22, 24, 47, 48, 65, 134]. Code diversification and randomization techniques (colloquially known as fine-grained ASLR [135]) can thwart code-reuse attacks by perturbing executable code at the function [12, 81], basic block [44, 87, 149], or instruction [72, 112] level so that the exact location of gadgets becomes *unpredictable* [92].

However, Snow et al. introduced JIT-ROP [135], a technique for bypassing fine-grained ASLR for 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 [10] 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 [9] and HideM [63] prevent code pages from being read by emulating the decades-old concept of *execute-only memory* (XOM) [31, 140] on contemporary architectures, like x86,[1] which lack native support for XOM. XnR marks code pages as "Not Present," resulting in a page fault (`#PF`) whenever an instruction fetch or data access is attempted on a code page. During 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. [46] and Conti et al. [28] showed that Oxymoron, XnR, and HideM can be bypassed using *indirect* JIT-ROP attacks by merely harvesting code pointers from (readable) data pages.

In response, Crane et al. [40, 41] introduced the concept of *leakage-resilient* diversification, which combines XOM and fine-grained ASLR with an indirect 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) [105] used in dynamic linking; trampolines are stored in XOM and cannot leak code layout. Readactor [40] is the first system to incorporate leakage-resilient code diversification. It layers CPH over a fine-grained ASLR scheme that leverages function permutation [12, 81] and instruction randomization [112], and implements XOM using a lightweight hypervisor.[2]

3 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 [5, 6, 129]. Attackers may overwrite kernel code pointers (e.g., function pointers, dispatch tables, return addresses) with *arbitrary* values [53, 138] through the interaction with the OS via buggy kernel interfaces. Examples include generic pseudo-filesystems (`procfs`, `debugfs` [33, 82]), the system call layer, and virtual device files (`devfs` [89]). Code pointers can be corrupted directly [53] or controlled indirectly (e.g., by first overwriting a pointer to a data structure that contains control data and subsequently tampering with its contents [54], in a manner similar to `vtable` pointer hijacking [130, 141]). Attackers may control *any* number of code pointers and trigger the kernel to dereference them on demand. Finally, we presume that the attackers are armed with an *arbitrary memory disclosure* bug [1, 4]. In particular, they may trigger the respective vulnerability *multiple* times, forcing the kernel to leak

[1] In x86 (both 32b and 64b), the execute permission implies read access.

[2] Readactor's hypervisor makes use of the Extended Page Tables (EPT) feature [61], 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.

the contents of *any* kernel-space memory address. Microarchitectural attacks, like Meltdown [99], Spectre [84], and similar side-channel attacks [68], are considered out of the scope of this article.

Hardening Assumptions. We assume an OS that implements the W^X policy [91, 97, 142] in kernel space. 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) [155], whereas for legacy systems we assume protection by KERNEXEC (PaX) [116] or kGuard [80]. In addition, we assume sane (read-only) memory permissions for the Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT) [30, 55]. Finally, the kernel may have support for kernel-space ASLR [50], stack-smashing protection [143], proper `.rodata` sections (constification of critical data structures) [142], pointer (symbol) hiding [128], SMAP/UDEREF [35, 115], page-table isolation (KPTI) [38, 69], or any other hardening feature. kR^X does not require or preclude any such features—they are orthogonal to our scheme(s). Data-only attacks, such as page table tampering [93] or process credentials modification [153], are considered out of the scope of this article; (self-)protecting such sensitive data structures [25, 43, 45] is also orthogonal to kR^X.

4 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 [2, 18, 129, 152, 153] in a ROP [73, 134] or JOP [24] fashion, or use other similar code-reuse techniques [22, 47, 48, 65, 144], including (in)direct JIT-ROP [28, 46, 135]. kR^X complements the work on user space leakage-resilient code diversification [19, 40] by providing a solution against code reuse for the *kernel setting*. The goal of kR^X is to aid commodity OS kernels in combatting (a) ROP/JOP and similar code-reuse attacks [47, 48, 65], (b) direct JIT-ROP, and (c) indirect JIT-ROP. To achieve that, it builds on two main pillars: (i) the R^X policy and (ii) fine-grained KASLR.

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 [40], HideM [63], and XnR [9]) 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 [62]) or paging nuances (e.g., `#PF`; XnR, TLB desynchronization; 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 [62, 94, 120, 127] or implemented using special-purpose hypervisors [40, 139, 147, 150]. 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, such as XOM, are enforced on selected applications by custom ancillary hypervisors [40]. Unfortunately, nested virtualization cripples scalability, as each nesting level results in ~6–8% of runtime overhead [11], excluding the additional overhead of the deployed protections.

The former approach is not impeccable either. Offloading security features (e.g., code integrity [127], XOM [62], and data integrity [147]) to commodity VMMs leads to a flat increase of 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 [148, 157],

some of which are implemented in superprivileged CPU modes [8, 157]. Considering the above and the fact that hypervisor exploits are becoming an indispensable part of the attackers' arsenal [59], 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 [62] or software executing in superprivileged CPU modes [8], (b) avoids (nesting) virtualization overheads, and (c) is on par with recent industry efforts [29]. Lastly, kR^X enables R^X capabilities even in systems that lack support for hardware-assisted virtualization.

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 similar 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 [40]) is geared toward the kernel setting. CPH was designed with support for C++, dynamic linking, and JIT compilation in mind. In contrast, commodity OSs (a) do not support C++ in kernel mode, hence `vtable` and exception handling, and COOP [131] 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 [57]; and (c) have limited support for JIT code in kernel space (typically to facilitate tracing and packet filtering [36]). These reasons prompted us to study new leakage-resilient diversification schemes, fine-tuned for the kernel.

5 DESIGN

5.1 R^X Enforcement

kR^X employs a self-protection approach to R^X, inspired by SFI [86, 106, 133, 146, 154]. 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 [106], NaCl [133, 154]) are designed for confining the control flow and memory-write operations of the sandboxed code, typically by imposing a canonical layout [133], bit-masking memory writes [146], and instrumenting computed branch instructions [106]. 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 [154].

In contrast, kR^X focuses on the *read* operations of benign code that can be abused to disclose memory [88]. (Memory reads are usually ignored by conventional SFI schemes owing to the non-trivial overhead associated with their instrumentation [19, 106].) 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) [76] to (almost) eliminate the protection overhead; (c) `kR^X-SEG`, a hardware-based R^X scheme that leverages memory segmentation (available in legacy systems) [75]; and (d) `kR^X-KAS`, a new kernel space layout that facilitates the efficient R^X enforcement by (a), (b), and (c).

5.1.1 `kR^X-KAS` *(x86 & x86-64).* The x86-64 architecture uses 48b virtual addresses that are sign-extended to 64b (bits `[48:63]` are copies of bit `[47]`), splitting the 64-b 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 1(a)) [83]:

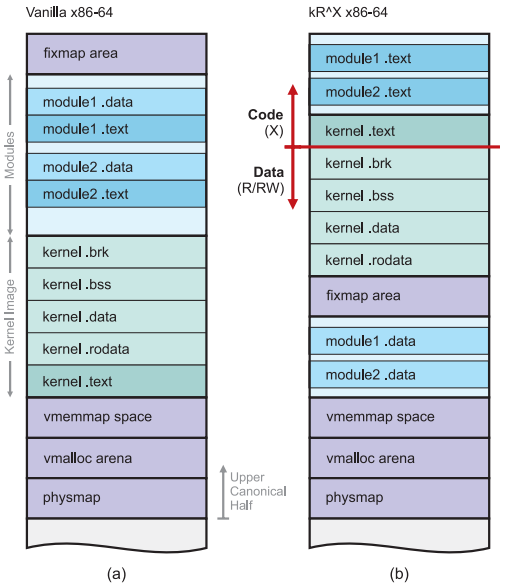

Fig. 1. The Linux kernel space layout in x86-64: (a) vanilla and (b) kR^X-KAS. The kernel image and modules regions may contain additional (ELF) sections; only the standard ones are shown.

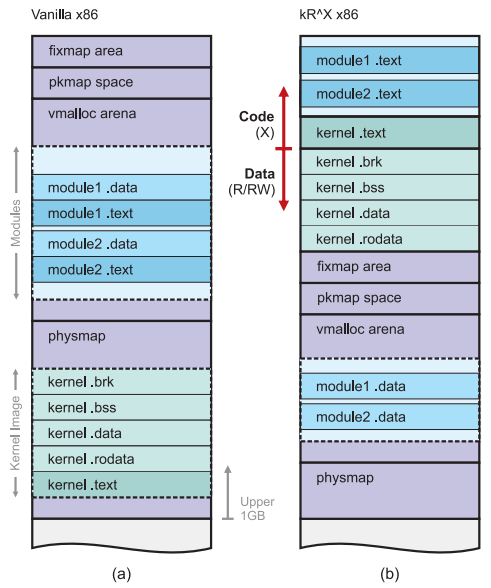

Fig. 2. The Linux kernel space layout in x86 (under the default 3G/1G user/kernel split): (a) vanilla and (b) kR^X-KAS. The kernel image and modules regions may contain extra sections.

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 32b platforms, different regions collide to prevent waste (e.g., kernel image and physmap, modules and vmalloc arena; see Figure 2(a)) [79].

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 1(b), x86-64, and Figure 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" toward 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.[3]

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 [16]. 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

[3]The default setting was selected by dividing the original modules area into two equally sized parts.

up in the data region.[4] The symbols _krx_edata and _text denote the end of the data region and the beginning of the code region in kR^X-KAS.

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 modules_text; 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 [17]).

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 [79]. Hence, as physical memory is allotted to the kernel image and modules, the existence of physmap results in *address aliasing*; virtual-address aliases, or *synonyms* [85], 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 [136]).

Alternative Layouts. kR^X-KAS has several advantages over the address space layouts imposed by SFI-based schemes (e.g., NaCl [154], LR^2 [19]). 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 32b systems, a smaller kernel space would necessitate the use of kmap/kunmap operations for managing page frames that cannot be directly addressed through physmap [79],[5] which, in turn, translates to higher runtime overhead. kmap/kunmap operations require altering the kernel page table, resulting in TLB pressure [109] and shootdowns. Second, the use of bit-masking confinement (similarly to NaCl [154] and LR^2 [19]) in the kernel setting requires a radically different set of memory allocators to cope with the alignment constraints of bit-masking. In contrast, the layout of kR^X-KAS is transparent to the kernel's performance-critical allocators [15]. Third, important kernel features that are tightly coupled with the kernel address space, such as KASLR [50] or alternative user/kernel splits (e.g., 2G/2G, 1G/3G) [32], 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 [56]. 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 32b. 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 (one security related) [124].

5.1.2 kR^X-SFI *(x86-64).* kR^X-SFI is a software-only R^X scheme that targets modern (64b) platforms. Once the kR^X-KAS layout is in place, R^X can be enforced by checking *all* memory reads and making sure that they fall within the data region (addresses below _krx_edata). As bit-masking load instructions is not an option owing to the non-canonical layout, kR^X-SFI employs

[4]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.

[5]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.

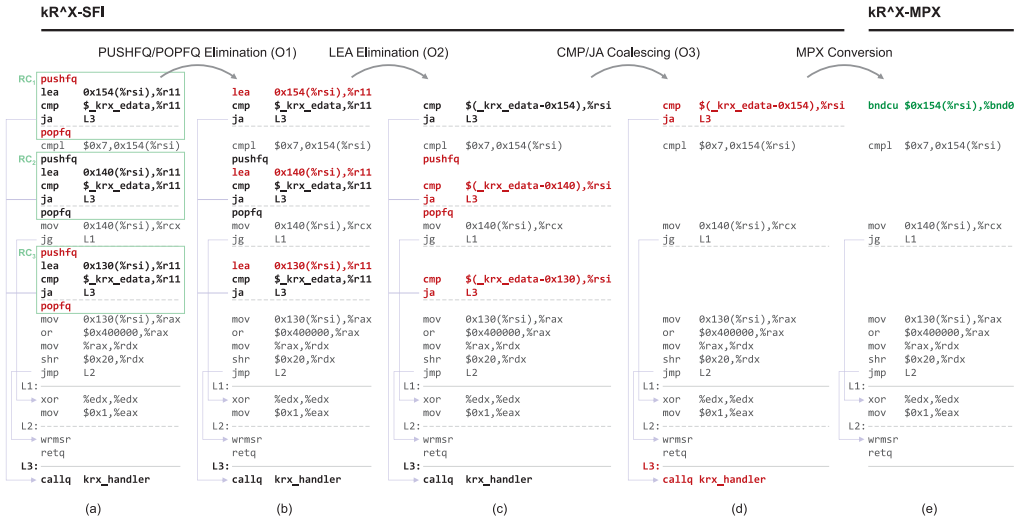

Fig. 3. The different optimization phases of kR^X-SFI (a)–(d) and kR^X-MPX (e).

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 to present the internals of kR^X-SFI. The original code excerpt is listed in Figure 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) [98]. 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 [106, 133, 146], but earlier work focuses on RISC-based architectures [19, 146] or fine-tunes bit-masking confinement [106]. 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.

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(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 an 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 such as *active kernel exploit response* [67] 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.

pushfq/popfq Elimination (O1). Spilling and filling the %rflags register is expensive [104]. However, we can eliminate *redundant* pushfq-popfq pairs by performing a liveness analysis on %rflags. Figure 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(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 7.2).[6]

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(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.

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 ($\texttt{offset}_a \texttt{!=} \texttt{offset}_b$), we can *coalesce* them to one RC that checks against the maximum displacement if in all control paths between RC_a and RC_b `%reg` is never (a) redefined and (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(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 that we require `%rsi` not to be spilled is to prevent temporal attacks, like those demonstrated by Conti et al. [28]. About one out of every two RCs can be eliminated using RC coalescing.

Stack Reads. If the stack pointer (`%rsp`) is used with a scaled index register [75], 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 that 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 (i.e., `.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.

String Operations and Safe Reads. The x86 string operations [75]—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

[6]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 overpreserve).

rep-prefixed, the RC is placed *after* the confined instruction, checking %rsi (or %rdi) once the respective operation is complete.[7] 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 owing to W^X. Safe reads account for 4% of all memory reads.

5.1.3 kR^X-MPX *(x86-64).* kR^X-MPX is a hardware-assisted R^X scheme that takes advantage of the MPX feature [76], 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.

MPX introduces four new bounds registers (%bnd0–%bnd3), each consisting of two 64b 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 an 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(e) illustrates the instrumentation performed by kR^X-MPX. Note that bndcu does not alter %rflags; thus, 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. 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.

5.1.4 kR^X-SEG *(x86).* In legacy (32b) systems, kR^X-SEG enforces the R^X policy using memory segmentation [75]. Note that the use of segmentation for isolation purposes has been well researched, both in user space [154] and kernel space [114] settings. Nevertheless, we 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 32b address space (4GB), neutralizing its effect. kR^X-SEG redefines the kernel data segment(s) to be on 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 used only by the stack-smashing protector [121, 143] and is limited to 4B (by default), whereas SS is left flat as well because of .krx_phantom (see "Stack Reads" in Section 5.1.2). Note that, in contrast to kR^X-{SFI, MPX}, kR^X-SEG enforces the R^X policy without relying on (kernel) code instrumentation.

5.2 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 [28,

[7] 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.

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

46]. The use of code perturbation or randomization to hinder code-reuse attacks has been studied extensively in the past [12, 44, 64, 72, 81, 87, 112, 149]. Previous research, however, either did not consider resilience to indirect JIT-ROP [28, 46] or focused on schemes geared toward userland code [19, 40]. kR^X introduces code diversification designed from the ground up to mitigate both *direct* and *indirect* (JIT-)ROP/JOP attacks for the kernel setting.

5.2.1 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 [44, 149]; at the section (`.text`) level, we perform function permutation [12, 81].

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 [44]. 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 [44, 149], and we augmented kR^X to resolve it as follows.

Starting with `k`, the number of randomization entropy bits *per function* that 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 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.

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 reuse only 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, 105]. Consequently, as we further discuss in Section 7.3, attackers must first use gadgets to initialize the appropriate registers before invoking a function.

5.2.2 Return-Address Protection. Return addresses are stored in kernel stacks, which are allocated from the readable data (`physmap`) region of `kR^X-KAS` [79]. Conti et al. demonstrated an indirect JIT-ROP attack that relies on harvesting return addresses from stacks [28]. kR^X treats return addresses specially to mitigate such indirect JIT-ROP attempts.

Return-Address Encryption (`X`). We employ an `XOR`-based encryption scheme to protect saved return addresses from being disclosed [19, 117, 151]. 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`

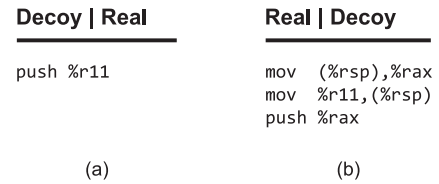

Fig. 4. Instrumentation code (function prologue; x86-64) to place the decoy return address (a) below or (b) above the real one.

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 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.

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 7.2. 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 [39].

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 and `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 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

[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).

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 [22, 47, 65] are coupled with a *pair* of return addresses (`R` and `D`), thereby forcing the attacker to randomly choose one of them. If `ncall`-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 $P_{succ} = 1/2^n$.

5.3 Limitations

Race Hazards. Both schemes presented in Section 5.2.2 obfuscate return addresses *after* they have been pushed (in cleartext) in the stack. Although this approach entails changes only at the callee side, it leaves a window open for an attacker to probe the stack and leak unencrypted/real return addresses [28]. In order for attackers to trigger the information disclosure bug, they need to interact with the OS via a kernel-exposed interface (see Section 3). Hence, they have to *surgically* time the execution of 1–3 kR^X instructions, with (a) process scheduling (which cannot be completely controlled, as it is affected by the runtime behavior of other processes on the system), (b) the cache/TLB side-effects of a CPU mode switch, and (c) the execution of the code required to trigger the leak—the latter can be up to thousands of instructions. We plan to further investigate this issue as part of our future work.

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: $[RS_1 \hat{} xkey_f]$. Likewise, if `f` is invoked from CS_2, RS_2 will be saved as $[RS_2 \hat{} xkey_f]$. Hence, if an attacker manages to leak both "ciphertexts," though the attacker cannot recover RS_1, RS_2, or $xkey_f$, the attacker may replace $[RS_1 \hat{} xkey_f]$ with $[RS_2 \hat{} xkey_f]$ (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 $[RS_1 \hat{} xkey_f]$ or $[RS_2 \hat{} xkey_f]$ with *any* harvested (encrypted) return address—say, $[RS_n \hat{} xkey_{f'}]$—is not a viable strategy because the respective return sites (RS_1/RS_2, RS_n) are encrypted with different keys ($xkey_f$, $xkey_{f'}$). Under return address encryption (`X`), substitution attacks are possible only among return addresses encrypted with the same `xkey`.

Substitution attacks resemble the techniques for overcoming coarse-grained CFI by stitching together `call`-preceded gadgets [22, 47, 65]. 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 be redirected only to the *valid* return sites of `f`, 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 of our return address protection schemes aim at thwarting JIT-ROP and, therefore, are not geared toward ensuring the integrity of code pointers [90]. In any case, they can be easily complemented with a register randomization scheme [40, 112], which foils `call`-preceded gadget chaining [19].

6 IMPLEMENTATION

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 Sections 5.1.2 and 5.1.3) and (b) randomize code blocks and protect return addresses (see

Sections 5.2.1 and 5.2.2). Our prototype consists of two plugins, krx 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 vartrackRTL optimization pass by calling GCC's register_callback() function and hooking with the pass manager [80]. 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 in 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 [28].

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 5.2.2) 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.

Kernel Support. kR^X-KAS (see Section 5.1.1) and kR^X-SEG (see Section 5.1.4) 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); and (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 supports *mixed* code: that is, both protected and unprotected modules. This design not only allows for incremental deployment and adoption but also facilitates selective hardening [60].

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 Sections 5.1 and 5.2 can all be incorporated in the assembler instead of the compiler, as they do not depend on high-level semantics.

Legitimate Code Reads. Kernel tracing and debugging (sub)systems such as ftrace and KProbes [36], 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.

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 [34] and (b) live kernel patching [37]. To provide support for the former, the BPF JIT compiler needs to be extended to include the techniques presented in Sections 5.1 and 5.2 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, and the like (see "Legitimate Code Reads," above).

7 EVALUATION

We studied the runtime overhead of kR^X-{SFI, MPX, SEG}, both as stand-alone implementations and when applied in conjunction with the code randomization schemes described in Section 5.2 (i.e., fine-grained KASLR coupled with return-address encryption or return-address decoys). We used the LMBench suite [107] for micro-benchmarking and employed the Phoronix Test Suite (PTS) [126] to measure the performance impact on real-world applications. (Note that the PTS is used by the Linux kernel developers to track performance regressions.) The reported results are average values of ten and 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.

7.1 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.

7.2 Performance

Micro-benchmarks. To assess the impact of kR^X on the various kernel subsystems and services, we used LMBench [107], focusing on two metrics: *latency* and *bandwidth* overhead. Specifically, we measured the additional latency imposed on (a) critical system calls, such as `open()`/`close()`, `read()`/`write()`, `select()`, `fstat()`, and `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()`, and `fork()+/bin/sh`); (d) signal installation (via `sigaction()`) and delivery; (e) protection faults and page faults; and (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 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)` to `SFI(-O3)` illustrate the effect of `pushfq`/`popfq` elimination, `lea` elimination, and `cmp`/`ja` coalescing when applied in 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. The last four columns (`SFI+D`, `SFI+X`, `MPX+D`, and `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

Table 1. kR^X Runtime Overhead on the LMBench Micro-benchmark (% over Vanilla Linux; x86-64)

	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%

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.

Table 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); 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%). 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;

Table 2. kR^X Runtime Overhead on the LMBench Micro-benchmark (% over Vanilla Linux; x86)

	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%

open()/close(), read()/write(), fork()+execve(), select (100 TCP fds), and pipe and socket I/O suffer the most.

Macro-benchmarks. To gain a better understanding of the performance implications of kR^X on realistic conditions, we used PTS [126]. PTS offers a number of *system* tests, such as ApacheBench, DBench, and IOzone, along with real-world workloads, such as extracting and building the Linux kernel. Table 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 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 of code diversification (i.e., fine-grained KASLR plus return-address decoys or return-address encryption) ranges between 0% and 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 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 two approaches, for SFI this is in favor of X (encryption) while for MPX it is in favor of D (decoys).

Table 3. kR^X Runtime Overhead on the Phoronix Test Suite
(% over Vanilla Linux; x86-64)

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 4. kR^X Runtime Overhead on the Phoronix Test Suite
(% over Vanilla Linux; x86)

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%**

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 use 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.

7.3 Security

Direct ROP/JOP. To assess the effectiveness of kR^X against direct ROP/JOP attacks, we used the ROP exploit for CVE-2013-2094 [3], 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 $P_{succ} = 1/2^{30}$, which we consider to be extremely low.

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.

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. Owing 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 [22, 47, 65]. 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 $P_{succ} = 1/2^n$.

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 (see Section 5.3), owing to function permutation, their leakage does not reveal anything about the immediate surrounding area of the disclosed routine. In addition, owing 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 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 30b, 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, 105]. This necessitates the use of gadgets for loading registers with the proper values. In essence, kR^X effectively restricts the attacker to data-only types of attacks on function pointers [137] (e.g., overwriting function pointers with the addresses of functions of the same, or lower, arity [51]).

[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.

8 RELATED WORK

We have already covered related work in the broader areas of code diversification and XOM in Sections 1 and 2. Most of these efforts are geared toward userland applications. As we discussed in Section 4, they are not quintessential for the OS kernel—especially when it comes to protecting against JIT-ROP. The main reasons include reliance on hypervisors [40, 41] (i.e., non-self-protection), secrecy of segment descriptors [10, 103], and custom page fault handling [9, 63] (i.e., non-realistic assumptions for the kernel setting), as well as designs that treat the kernel as part of the TCB. In addition, some of the proposed schemes suffer from high overheads, which are prohibitive for the OS kernel [46].

LR^2 [19] and KHide [62] are two previously proposed systems that are closer to (some aspects of) kR^X. LR^2 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. As discussed in Section 5.1.1 ("Alternative Layouts"), 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 constraints) and result in a whopping address space waste (e.g., LR^2 squanders half of the address space). At the same time, kR^X (a) focuses on a different architecture and domain (x86 vs. 32b ARM, kernel vs. user space), (b) can leverage hardware support when available (segmentation on x86 and MPX on x86-64), and (c) is readily compatible with modern Linux distributions without requiring modifications to existing applications (in contrast to LR^2's `glibc` compatibility issues). KHide, similarly to kR^X, protects the OS kernel against code0reuse attacks, but relies on a commodity VMM (KVM) to do so; kR^X adopts a self-protection-based approach instead. More importantly, KHide does not conceal return addresses, which is important for defending against indirect JIT-ROP attacks [28].

SECRET [158] provides XOM-equivalent protection to COTS binaries, using memory segmentation on x86 and information hiding on x86-64, while NORAX [27] leverages a combination of MMU permission bits to retrofit XOM to ARM binaries. In contrast, kR^X enforces XOM on architectures that lack native support for marking memory pages as execute-only and employs strong memory isolation mechanisms (`kR^X-{SFI, MPX, SEG}`), avoiding the use of information hiding to guard against direct JIT-ROP attacks, as this strategy has been shown to be ineffective in the kernel setting [70, 74, 78]. Furthermore, kR^X combines XOM with return-address protection and fine-grained KASLR, defending against any kind of attack that relies on pre-computed gadget addresses. Lastly, both SECRET and NORAX target userland applications, whereas kR^X is geared toward the OS kernel.

Live Re-randomization. Giuffrida et al. [64] introduced modifications to MINIX so that the system can be re-randomized periodically, at runtime. This is an orthogonal approach to kR^X, best suited for microkernels and not kernels with a monolithic design, while it incurs a significant runtime overhead for short re-randomization intervals. TASR [13] 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 [151] and CodeArmor [26] re-randomize userland applications continuously, treating the OS kernel as part of their TCB. Lastly, RuntimeASLR [101] 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 [42] augments FreeBSD with support for coarse-grained CFI, whereas Fine-CFI [94] and the system presented by Ge et al. [58] rectify the enforcement approach of HyperSafe [148] 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 [117] 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 [21, 47, 51, 65].

Heisenbyte [139] and NEAR [150] employ destructive code reads to thwart attacks that rely on code disclosures (e.g., JIT-ROP). Alas, Snow et al. [136] demonstrated that destructive code reads can be undermined with code inference attacks. More recently, Pewny et al. [123] 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, a design choice that is not ideal for the kernel setting, as we describe in Section 4.

Li et al. [95] 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 [24, 65]. kR^X, on the other hand, provides comprehensive protection against all types of (known) code-reuse attacks. Chen et al. [25] 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. PrivWatcher assumes that the kernel is not vulnerable to code reuse attacks and is therefore orthogonal to kR^X. Song et al. proposed KENALI [137] to defend against data-only attacks. KENALI enforces kernel dataflow integrity [23] 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. [96] 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 recreates complex OS functionality using only popular paths; similarly to KENALI, the overhead of Lind is also very high (up to 525%).

9 CONCLUSION

As the complete eradication of kernel memory corruption and disclosure vulnerabilities remains a challenging task, defenses against their exploitation become imperative. In this article, we investigated a previously unexplored point in the design space by presenting kR^X: a practical hardening scheme that fulfills the current lack of self-protection-based, execute-only kernel memory. Implemented as a GCC plugin and a set of kernel patches, kR^X is readily applicable on x86(-64) Linux, it does not rely on a hypervisor or any other more privileged entity, it does not require modifications to existing applications, and it incurs a low runtime overhead, benefiting from the availability of MPX and memory segmentation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] 2010. CVE-2010-3437.
[2] 2011. Analysis of jailbreakme v3 font exploit. Retrieved November 24, 2018 from https://goo.gl/RGsgzc.
[3] 2013. CVE-2013-2094.
[4] 2013. CVE-2013-6282.
[5] 2015. CVE-2015-3036.
[6] 2015. CVE-2015-3290.

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proc. ACM CCS*. 340–353.
[8] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the ARM TrustZone secure world. In *Proc. ACM CCS*. 90–102.
[9] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. 2014. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proc. of ACM CCS*. 1342–1353.
[10] Michael Backes and Stefan Nürnberger. 2014. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *Proc. USENIX Sec.* 433–447.
[11] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The turtles project: Design and implementation of nested virtualization. In *Proc. USENIX OSDI*. 423–436.
[12] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. USENIX Sec.* 255–270.
[13] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely rerandomization for mitigating memory disclosures. In *Proc. ACM CCS*. 268–279.
[14] Andrea Bittau. 2013. Linux Kernel < 3.8.9 (x86_64) 'perf_swevent_init' Privilege Escalation. Retrieved November 24, 2018 from https://www.exploit-db.com/exploits/26131/.
[15] Jeff Bonwick. 1994. The slab allocator: An object-caching kernel memory allocator. In *Proc. of USENIX Summer*. 87–98.
[16] Daniel Pierre Bovet. 2013. Special sections in Linux binaries. Retrieved November 24, 2018 from https://lwn.net/Articles/531148/.
[17] Daniel P. Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media, Chapter Modules, 842–851.
[18] Brad Spengler and Sorbo. 2014. Linux perf_swevent_init Privilege Escalation. Retrieved November 24, 2018 from https://goo.gl/eLgE48.
[19] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2016. Leakage-resilient layout randomization for mobile devices. In *Proc. NDSS*.
[20] Adrian Bunk. 2006. i386: always enable regparm. Retrieved November 24, 2018 from https://goo.gl/uo6taH.
[21] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. USENIX Sec.* 161–176.
[22] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *Proc. USENIX Sec.* 385–399.
[23] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *Proc. of USENIX OSDI*. 147–160.
[24] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proc. ACM CCS*. 559–572.
[25] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. 2017. PrivWatcher: Non-bypassable monitoring and protection of process credentials from memory corruption attacks. In *Proc. ASIACCS*. 167–178.
[26] Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. CodeArmor: Virtualizing the code space to counter disclosure attacks. In *Proc. IEEE EuroS&P*. 514–529.
[27] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. 2017. NORAX: Enabling execute-only memory for COTS binaries on AArch64. In *Proc. IEEE S&P*. 304–319.
[28] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, Marco Negro, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proc. ACM CCS*. 952–963.
[29] Kees Cook. [n.d.]. Kernel Self Protection Project. Retrieved November 24, 2018 from https://goo.gl/KsN0t8.
[30] Kees Cook. 2013. x86: make IDT read-only. Retrieved November 24, 2018 from https://lkml.org/lkml/2013/4/8/749.
[31] F. J. Corbató and V. A. Vyssotsky. 1965. Introduction and overview of the multics system. In *Proc. AFIPS*. 185–196.
[32] Jonathan Corbet. 2004. Virtual Memory I: the problem. Retrieved November 24, 2018 from http://lwn.net/Articles/75174/.
[33] Jonathan Corbet. 2009. An updated guide to debugfs. Retrieved November 24, 2018 from https://lwn.net/Articles/334546/.
[34] Jonathan Corbet. 2011. A JIT for packet filters. Retrieved November 24, 2018 from https://lwn.net/Articles/437981/.
[35] Jonathan Corbet. 2012. Supervisor mode access prevention. Retrieved November 24, 2018 from https://lwn.net/Articles/517475/.

[36] Jonathan Corbet. 2014. BPF: the universal in-kernel virtual machine. Retrieved November 24, 2018 from https://lwn.net/Articles/599755/.
[37] Jonathan Corbet. 2015. A rough patch for live patching. Retrieved November 24, 2018 from https://lwn.net/Articles/634649/.
[38] Jonathan Corbet. 2017. Retrieved November 24, 2018 from The current state of kernel page-table isolation. https://lwn.net/Articles/741878/.
[39] Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Booby trapping software. In *Proc. NSPW*. 95–106.
[40] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical code randomization resilient to memory disclosure. In *Proc. IEEE S&P*. 763–780.
[41] Stephen J. Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's a TRaP: Table randomization and protection against function-reuse attacks. In *Proc. ACM CCS*. 243–255.
[42] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *Proc. IEEE S&P*. 292–307.
[43] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. 2015. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proc. ACM ASPLOS*. 191–206.
[44] Lucas Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2013. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *Proc. ACM ASIACCS*. 299–310.
[45] Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. PT-rand: Practical mitigation of data-only attacks against page tables. In *Proc. NDSS*.
[46] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *Proc. NDSS*.
[47] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proc. USENIX Sec.* 401–416.
[48] Solar Designer. 1997. Getting around non-executable stack (and fix). Retrieved November 24, 2018 from http://seclists.org/bugtraq/1997/Aug/63.
[49] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *Proc. USENIX Sec.* 131–148.
[50] Jake Edge. 2013. Kernel address space layout randomization. Retrieved November 24, 2018 from https://lwn.net/Articles/569635/.
[51] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proc. ACM CCS*. 901–913.
[52] Exploit Database. 2012. EBD-20201.
[53] Exploit Database. 2014. EBD-31346.
[54] Exploit Database. 2014. EBD-33516.
[55] Thomas Garnier. 2017. 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.
[56] GCC online documentation. [n.d.]. Intel 386 and AMD x86-64 Options. Retrieved November 24, 2018 from https://goo.gl/38gK86.
[57] Xinyang Ge, Mathias Payer, and Trent Jaeger. 2017. An evil copy: How the loader betrays you. In *Proc. of NDSS*.
[58] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *Proc. IEEE EuroS&P*. 179–194.
[59] Jason Geffner. 2015. VENOM: Virtualized Environment Neglected Operations Manipulation. Retrieved November 24, 2018 from http://venom.crowdstrike.com.
[60] Dimitris Geneiatakis, Georgios Portokalidis, Vasileios P. Kemerlis, and Angelos D. Keromytis. 2012. Adaptive defenses for commodity software through virtual application partitioning. In *Proc. CCS*. 133–144.
[61] Matthew Gillespie. 2015. Best practices for Pparavirtualization enhancements from Intel® virtualization Ttechnology: EPT and VT-d. Retrieved November 24, 2018 from https://goo.gl/LLlAZK.
[62] Jason Gionta, William Enck, and Per Larsen. 2016. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Proc. IEEE CNS*. 189–197.
[63] Jason Gionta, William Enck, and Peng Ning. 2015. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proc. ACM CODASPY*. 325–336.
[64] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proc. USENIX Sec.* 475–490.

[65] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *Proc. IEEE S&P*. 575–589.
[66] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. 2014. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proc. of USENIX Sec.* 417–432.
[67] grsecurity. 2011. Active kernel exploit response. Retrieved November 24, 2018 from https://xorl.wordpress.com/2011/04/27/grkernsec_kern_lockout-active-kernel-exploit-response/.
[68] Daniel Gruss. 2017. *Software-based Microarchitectural Attacks*. Ph.D. Dissertation. Graz University of Technology.
[69] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is dead: Long live KASLR. In *Proc. ESSoS*. 161–176.
[70] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proc. ACM CCS*. 368–379.
[71] Dave Hansen. 2015. [RFC] x86: Memory Protection Keys. Retrieved November 24, 2018 from https://lwn.net/Articles/643617/.
[72] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. 2012. ILR: Where'd my gadgets go?. In *Proc. IEEE S&P*. 571–585.
[73] Ralf Hund, Thorsten Holz, and Felix C. Freiling. 2009. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *Proc.USENIX Sec.* 384–398.
[74] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Proc. IEEE S&P*. 191–205.
[75] Intel Corporation. 2015. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. https://software.intel.com/en-us/articles/intel-sdm.
[76] Intel Corporation. 2016. *Intel® Memory Protection Extensions Enabling Guide*. https://software.intel.com/sites/default/files/managed/9d/f6/Intel_MPX_EnablingGuide.pdf.
[77] Intel® OS Guard (SMEP). 2013. Intel® Xeon® Processor E5-2600 V2 Product Family Technical Overview. Retrieved November 24, 2018 from https://goo.gl/mS5Ile.
[78] Yeongjin Jang, Sangho Lee, and Taesoo Kim. 2016. Breaking kernel address space layout randomization with Intel TSX. In *Proc. oACM CCS*. 380–392.
[79] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *Proc. USENIX Sec.* 957–972.
[80] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight kernel protection against return-to-user attacks. In *Proc. of USENIX Sec.* 459–474.
[81] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. 2006. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proc. ACSAC*. 339–348.
[82] Thomas J. Killian. 1984. Processes as files. In *Proc. of USENIX Summer*. 203–207.
[83] Andi Kleen. 2004. Memory Layout on amd64 Linux. Retrieved November 24, 2018 from https://goo.gl/BtvguP.
[84] 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. 2019. Spectre attacks: Exploiting speculative execution. *To Appear in Proc. IEEE S&P* (May 2019).
[85] Eric J. Koldinger, Jeffrey S. Chase, and Susan J. Eggers. 1992. Architecture support for single address space operating systems. In *Proc. ACM ASPLOS*. 175–186.
[86] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *Proc. EuroSys*. 437–452.
[87] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Proc. IEEE S&P*. 472–488.
[88] Mathias Krause. 2013. CVE Requests (maybe): Linux kernel: various info leaks, some `NULL` ptr derefs. Retrieved November 24, 2018 from http://www.openwall.com/lists/oss-security/2013/03/05/13.
[89] Greg Kroah-Hartman. 2003. `udev` – A userspace implementation of devfs. In *Proc. OLS*. 263–271.
[90] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer integrity. In *Proc. USENIX OSDI*. 147–163.
[91] Mike Larkin. 2015. Kernel W^X improvements in OpenBSD. In *Hackfest*.
[92] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated software diversity. In *Proc. IEEE S&P*. 276–291.
[93] JungSeung Lee, HyoungMin Ham, InHwan Kim, and JooSeok Song. 2015. POSTER: Page table manipulation attack. In *Proc. ACM CCS*. 1644–1646.
[94] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. 2018. Fine-CFI: Fine-grained control-flow integrity for operating system kernels. *IEEE Trans. Inf. Forensics Security* 13, 6 (June 2018), 1535–1550.

[95] Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, and Sina Bahram. 2010. Defeating return-oriented rootkits with "*return-less*" kernels. In *Proc. EuroSys*. 195–208.
[96] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proc. ATC*. 1–13.
[97] Siarhei Liakh. 2009. NX protection for kernel data. Retrieved November 24, 2018 from https://lwn.net/Articles/342266/.
[98] Linux Cross Reference. [n.d.]. Linux kernel release 3.19. Retrieved November 24, 2018 from http://lxr.free-electrons.com/source/arch/x86/kernel/cpu/perf_event_intel_uncore_snb.c?v=3.19#L565.
[99] 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. 2018. Meltdown: Reading kernel memory from user space. In *Proc. USENIX Sec.* 973–990.
[100] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proc. ACM CCS*. 1607–1619.
[101] Kangjie Lu, Stefan Nürnberger, Michael Backes, and Wenke Lee. 2016. How to make ASLR win the clone wars: Runtime re-randomization. In *Proc. NDSS*.
[102] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive kernel memory initialization to eliminate data leakages. In *Proc. ACM CCS*. 920–932.
[103] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-guard: Stopping address space leakage for code reuse attacks. In *Proc. ACM CCS*. 280–291.
[104] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM PLDI*. 190–200.
[105] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2013. System V Application Binary Interface. Retrieved November 24, 2018 from http://www.x86-64.org/documentation/abi.pdf.
[106] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC architecture. In *Proc. USENIX Sec.* 209–224.
[107] Larry McVoy and Carl Staelin. 1996. lmbench: Portable tools for performance analysis. In *Proc. USENIX ATC*. 279–294.
[108] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and practical mitigation of uninitialized read vulnerabilities. In *Proc. NDSS*.
[109] Ingo Molnar. 2003. 4G/4G split on x86, 64 GB RAM (and more) support. Retrieved November 24, 2018 from http://lwn.net/Articles/39283/.
[110] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *Proc. ACM PLDI*. 577–587.
[111] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proc. ACM CCS*. 914–926.
[112] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proc. IEEE S&P*. 601–615.
[113] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2013. Transparent ROP exploit mitigation using indirect branch tracing. In *Proc. USENIX Sec.* 447–462.
[114] PaX Team. 2007. UDEREF/i386. Retrieved November 24, 2018 from http://grsecurity.net/ spender/uderef.txt.
[115] PaX Team. 2010. UDEREF/amd64. Retrieved November 24, 2018 from https://goo.gl/iPuOVZ.
[116] PaX Team. 2011. Better kernels with GCC plugins. Retrieved November 24, 2018 from https://lwn.net/Articles/461811/.
[117] PaX Team. 2015. RAP: RIP ROP. In *Hackers 2 Hackers Conference (H2HC)*.
[118] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *Proc. DIMVA*. 144–164.
[119] Enrico Perla and Massimiliano Oldani. 2010. Stairway to successful kernal Exploitation. In *A Guide To Kernel Exploitation: Attacking the Core*. Elsevier. 47–99.
[120] Nick L. Petroni, Jr. and Michael Hicks. 2007. Automated detection of persistent kernel control-flow attacks. In *Proc. ACM CCS*. 103–115.
[121] Theofilos Petsios, Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2015. DynaGuard: Armoring canary-based protections against brute-force attacks. In *Proc. ACSAC*. 351–360.
[122] Jannik Pewny and Thorsten Holz. 2013. Control-flow restrictor: Compiler-based CFI for iOS. In *Proc. ACSAC*. 309–318.
[123] Jannik Pewny, Philipp Koppe, Lucas Davi, and Thorsten Holz. 2017. Breaking and fixing destructive code read defenses. In *Proc. ACSAC*. 55–67.
[124] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive kernel protection against just-in-time code reuse. In *Proc. EuroSys*. 420–436.

[125] Gerald J. Popek and David A. Farber. 1978. A model for verification of data security in operating systems. *Commun. ACM* 21, 9 (September 1978), 737–749.
[126] PTS. [n.d.]. Phoronix Test Suite. Retrieved November 24, 2018 from http://www.phoronix-test-suite.com.
[127] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing. In *Proc. RAID*. 1–20.
[128] Dan Rosenberg. 2010. `kptr_restrict` for hiding kernel pointers. Retrieved November 24, 2018 from https://lwn.net/Articles/420403/.
[129] Chris Salls. 2017. Linux Kernel 4.13 (Ubuntu 17.10) - '`waitid()`' SMEP/SMAP/Chrome Sandbox Privilege Escalation. Retrieved November 24, 2018 from https://www.exploit-db.com/exploits/43127/.
[130] Pawel Sarbinowski, Vasileios P. Kemerlis, Cristiano Giuffrida, and Elias Athanasopoulos. 2016. VTPin: Practical VTable hijacking protection for binaries. In *Proc. ACSAC*. 448–459.
[131] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *Proc. IEEE S&P*. 745–762.
[132] SecurityFocus. 2009. Linux Kernel 'perf_counter_open()' Local Buffer Overflow Vulnerability. https://www.securityfocus.com/bid/36423/.
[133] David Sehr, Robert Muth, Cliff L. Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. 2010. Adapting software fault isolation to contemporary CPU architectures. In *Proc. USENIX Sec.* 1–11.
[134] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proc. ACM CCS*. 552–61.
[135] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proc. IEEE S&P*. 574–588.
[136] Kevin Z. Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. 2016. Return to the zombie gadgets: Undermining destructive code reads via code inference attacks. In *Proc. IEEE S&P*. 954–968.
[137] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. 2016. Enforcing kernel security invariants with data flow integrity. In *Proc. NDSS*.
[138] Brad Spengler. 2014. Enlightenment Linux Kernel Exploitation Framework. Retrieved November 24, 2018 from https://goo.gl/hDymQg.
[139] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *Proc. ACM CCS*. 256–267.
[140] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. In *Proc. ACM ASPLOS*. 168–177.
[141] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proc. USENIX Sec.* 941–955.
[142] Arjan van de Ven. 2005. Debug option to write-protect rodata: the write protect logic and config option. Retrieved November 24, 2018 from https://goo.gl/shDf0o.
[143] Arjan van de Ven. 2006. Add `-fstack-protector` support to the kernel. Retrieved November 24, 2018 from https://lwn.net/Articles/193307/.
[144] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. 2017. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proc. ACM CCS*. 1675–1689.
[145] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. 2014. Dynamic hooks: Hiding control flow changes within non-control data. In *Proc. USENIX Sec.* 813–828.
[146] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient software-based fault isolation. In *Proc. ACM SOSP*. 203–216.
[147] Xiaoguang Wang, Yue Chen, Zhi Wang, Yong Qi, and Yajin Zhou. 2015. SecPod: A framework for virtualization-based security systems. In *Proc. USENIX ATC*. 347–360.
[148] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proc. IEEE S&P*. 380–395.
[149] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proc. ACM CCS*. 157–168.
[150] Jan Werner, George Baltas, Rob Dallara, Nathan Otternes, Kevin Snow, Fabian Monrose, and Michalis Polychronakis. [n.d.]. No-execute-after-read: Preventing code disclosure in commodity software. In *Proc. ACM ASIACCS*. 35–46.
[151] 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. 2016. Shuffler: Fast and deployable continuous code re-randomization. In *Proc. USENIX OSDI*. 367–382.

[152] Rafal Wojtczuk. 2015. Exploiting "BadIRET" vulnerability (CVE-2014-9322, Linux kernel privilege escalation). Retrieved November 24, 2018 from https://goo.gl/bSEhBI.
[153] Wen Xu and Yubin Fu. 2015. Own your Android! Yet another universal root. In *Proc. USENIX WOOT*.
[154] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE S&P*. 79–93.
[155] Fenghua Yu. 2011. Enable/disable supervisor mode execution protection. Retrieved November 24, 2018 from https://goo.gl/utKHno.
[156] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *Proc. IEEE S&P*. 559–573.
[157] Fengwei Zhang, Jiang Wang, Kun Sun, and Angelos Stavrou. 2014. HyperCheck: A hardware-assisted integrity monitor. *IEEE Transactions on Dependable and Secure Computing* 11, 4 (July/August 2014), 332–344.
[158] Mingwei Zhang, Michalis Polychronakis, and R. Sekar. 2017. Protecting COTS binaries from disclosure-guided code reuse attacks. In *Proc. ACSAC*. 128–140.
[159] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proc. USENIX Sec.* 337–352.

Received February 2018; revised June 2018; accepted September 2018