kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse

Marios Pomonis* Theofilos Petsios* Angelos D. Keromytis*
Michalis Polychronakis† Vasileios P. Kemerlis‡

*Columbia University {mpomonis, theofilos, angelos}@cs.columbia.edu
†Stony Brook University mikepo@cs.stonybrook.edu
‡Brown University vpk@cs.brown.edu

Abstract

The abundance of memory corruption and disclosure vulnerabilities in kernel code necessitates the deployment of hardening techniques to prevent privilege escalation attacks. As more strict memory isolation mechanisms between the kernel and user space, like Intel's SMEP, become commonplace, attackers increasingly rely on code reuse techniques to exploit kernel vulnerabilities. Contrary to similar attacks in more restrictive settings, such as 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 certain code snippets and 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 user-land software, but so far this approach has not been applied for the protection of the kernel itself.

In this paper, 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 the x86-64 Linux kernel and can benefit from hardware support (e.g., MPX on modern Intel CPUs) 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.

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

Keywords Execute-only memory, Code diversification

EuroSys '17, April 23–26, 2017, Belgrade, Serbia

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064216

1. Introduction

The deployment of standard kernel hardening schemes, like address space layout randomization (KASLR) [40] and non-executable memory [71], has prompted a shift from legacy code injection (in kernel space) to *return-to-user* (ret2usr) attacks [62]. 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/-data flow is hijacked and redirected to code/data residing in user space, effectively bypassing KASLR and (kernel-space) W^X. Fortunately, however, recent software [62, 90] and hardware [61] 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, like return-oriented programming (ROP) [104]; old ret2usr exploits [8] are converted to use ROP payloads instead of shellcode [18], while modern jailbreak and privilege escalation exploits rely solely on code-reuse [2, 121]. At the same time, the security community developed protections against code-reuse attacks: control-flow integrity (CFI) [7] and code diversification [38, 57, 87, 118] schemes have been applied both in the user as well as in the kernel setting [34, 53]. Unfortunately, these solutions are not bulletproof. Recent studies demonstrated that both coarse-grained [88, 125, 126] and fine-grained [85, 86, 92, 95, 111] CFI schemes can be bypassed by confining the hijacked control flow to valid execution paths [21, 36, 42, 54, 55], while code diversification can be circumvented by leveraging *memory disclosure* vulnerabilities [105].

Having the ability to disclose the memory contents of a process, exploit code can (dynamically at runtime) pinpoint the exact location of ROP gadgets, and assemble them on-the-fly into a functional ROP payload. This kind of "just-in-time" ROP (JIT-ROP) [105] is particularly effective against applications with integrated scripting support, like 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, has increased flexibility in mounting a JIT-ROP attack on a diversified kernel [53], as *any* user program may attack the OS. Marked, kernel JIT-ROP attacks are not only easier to mount, but are also facilitated by the abundance of memory disclosure vulnerabilities in kernel code [68, 77, 80].

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. Nevertheless, given that widely-used CPU architectures, like 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 [11], TLB desynchronization [51], hardware virtualization [32, 52], or techniques inspired by software-fault isolation (SFI) [19]. A common characteristic of these schemes (except LR^2 [19]) is that they rely on a more privileged domain (e.g., the OS kernel [11, 51] or a hypervisor [32, 52]) 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 more prevalent in the kernel setting, the need for an effective kernel defense against them becomes more imperative than ever.

Retrofitting existing hypervisor-based approaches for kernel protection can be an option [52], 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, thereby resulting in high runtime overheads [12]; second, when implemented as part of an existing hypervisor [52], 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 [123] and LR^2 [19], along with other design decisions that we discuss in detail 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 processors, is available to userland software only [56].

In this paper, we study a previously unexplored point in the design space continuum by presenting `kR^X`: a both *comprehensive* and *practical* kernel hardening solution that diversifies the kernel's code and prevents any read accesses to it. More importantly, the latter is achieved by following a *self-protection* approach that relies on code instrumentation to apply SFI-inspired 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 [64, 118], to thwart the direct use of pre-selected ROP gadgets; and ii) return address protection using either a XOR-based encryption scheme [19, 91, 120] or decoy return addresses, to thwart gadget inference through saved return addresses on the kernel stacks [24].

Practical applicability to existing systems is ensured given that kR^X: i) does not rely on more privileged entities (e.g., a hypervisor [32, 52]) than the kernel itself; ii) is readily applicable on x86-64 systems, and can leverage memory protection extensions (i.e., Intel's MPX [60]) 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 support is available.

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, both in terms of permissions and contents, part of the kernel-accessible memory (i.e., the user space part) [62]. In particular, they can execute code with kernel rights by hijacking a kernel control path and redirecting it to user space, effectively invalidating the protection(s) offered by standard defenses, like kernel-space ASLR [40] and W^X [71]. Attacks of this kind, known as *return-to-user (ret2usr)*, can be traced back to the early 1970's, as demonstrated by the PDP-10 "address wraparound" fault [96]. Over the past decade, however, ret2usr has been promoted to the de facto kernel exploitation technique [93].

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 [102], function pointers [108], and dispatch tables [43], because these facilitate code execution. Nevertheless, pointers to *critical data structures* stored in the kernel data section or heap (i.e., non-control data [114]) are also targets, as they enable attackers to tamper with the data contained in certain objects by mapping fake copies in user space [45]. The targeted data structures typically contain data that affect the control flow (e.g., code pointers), so as to diverge execution to arbitrary locations. The net result of all ret2usr attacks is that the control/data flow of the kernel is hijacked and redirected to user space code/data [62].

Code Reuse Prevention Code reuse exploits rely on code fragments (gadgets) located at *predetermined* memory addresses [20, 23, 36, 39, 54, 104]. Code diversification

and randomization techniques (colloquially known as fine-grained ASLR [105]) can thwart code-reuse attacks by perturbing executable code at the function [13, 64], basic block [38, 118], or instruction [57, 87] level, so that the exact location of gadgets becomes *unpredictable* [72].

However, Snow et al. introduced "just-in-time" ROP (JIT-ROP) [105], 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 [11] and HideM [51] prevent code pages from being read by emulating the decades-old concept of *execute-only memory* (XOM) [26, 110] on contemporary architectures, like x86,[1] which lack native support for XOM. XnR marks code pages as "Not Present," resulting into a page fault (`#PF`) whenever an instruction fetch or data access is attempted on a code page; upon such an event, the OS verifies the source of the fault and temporarily marks the page as present, readable and executable, or terminates execution. HideM leverages the fact that x86 has separate Translation Look-aside 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 (corresponding to code pages) map to different page frames depending on the TLB consulted. Alas, Davi et al. [37] and Conti et al. [24] showed that Oxymoron, XnR, and HideM can be bypassed using *indirect* JIT-ROP attacks by merely harvesting code pointers from (readable) data pages.

As a response, Crane et al. [32, 33] introduced the concept of *leakage-resilient* diversification, which combines XOM and fine-grained ASLR with an indirection mechanism called code-pointer hiding (CPH). Fine-grained ASLR and XOM foil direct (JIT-)ROP, whereas CPH mitigates indirect JIT-ROP by replacing code pointers in readable memory with pointers to arrays of direct jumps (trampolines) to function entry points and return sites—CPH resembles the Procedure Linkage Table (PLT) [82] used in dynamic linking; trampolines are stored in XOM and cannot leak code layout. Readactor [32] is the first system to incorporate leakage-resilient code diversification. It layers CPH over a fine-grained ASLR scheme that leverages function permutation [13, 64] and instruction randomization [87], and implements XOM using a lightweight hypervisor.[2]

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

[2] Readactor's hypervisor makes use of the Extended Page Tables (EPT) [50] feature 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.

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 elevate their privileges by exploiting kernel-memory corruption bugs [5, 6]. Attackers may overwrite kernel code pointers (e.g., function pointers, dispatch tables, return addresses) with *arbitrary* values [44, 108], through the interaction with the OS via buggy kernel interfaces. Examples include generic pseudo-filesystems (`procfs`, `debugfs` [27, 65]), the system call layer, and virtual device files (`devfs` [69]). Code pointers can be corrupted directly [44] or controlled indirectly (e.g., by first overwriting a pointer to a data structure that contains control data and subsequently tampering with its contents [45], in a manner similar to `vtable` pointer hijacking [100, 111]). Attackers may control *any* number of code pointers and trigger the kernel to dereference them on demand. (Note that this is not equivalent to an "arbitrary write" primitive.) Finally, we presume that the attackers are armed with an *arbitrary memory disclosure* bug [1, 4]. In particular, they may trigger the respective vulnerability *multiple* times, forcing the kernel to leak the contents of *any* kernel-space memory address.

Hardening Assumptions We assume an OS that implements the W^X policy [71, 75, 112] in kernel space.[3] 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) [124], whereas for legacy systems we assume protection by KERNEXEC (PaX) [90] or kGuard [62]. Finally, the kernel may have support for kernel-space ASLR [40], stack-smashing protection [113], proper `.rodata` sections (constification of critical data structures) [112], pointer (symbol) hiding [99], SMAP/UDEREF [28, 89], or any other hardening feature. kR^X does not require or preclude any such features; they are orthogonal to our scheme(s). Page table tampering [73] is considered out of scope; (self-)protecting page tables [35] is also orthogonal to kR^X and part of our ongoing work.

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, 121, 122] in a ROP [58, 104] or JOP [23] fashion, or use other similar code reuse techniques [20, 36, 39, 54], including (in)direct JIT-ROP [24, 37, 105].

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

kR^X complements the work on user space leakage-resilient code diversification [19, 32] by providing a solution against code reuse for the *kernel setting*. The goal of kR^X is to aid commodity OS kernels (simultaneously) combat: (a) ROP/JOP and similar code reuse attacks [36, 39, 54], (b) direct JIT-ROP, and (c) indirect JIT-ROP. To achieve that, it builds upon two main pillars: (i) the R^X policy, and (ii) fine-grained KASLR.

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 [32], HideM [51], XnR [11]) 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 [52]) 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 [52, 94, 98] or implemented using special-purpose hypervisors [32, 109, 116, 119]. The latter result in a smaller trusted computing base (TCB), but they typically require *nesting* hypervisors to attain comprehensive protection. Note that nesting occurs naturally in cloud settings, where contemporary (infrastructure) VMMs are in place and offbeat security features, like XOM, are enforced on selected applications by custom, ancillary hypervisors [32]. Alas, nested virtualization cripples scalability, as each nesting level results in ∼6–8% of runtime overhead [12], excluding the additional overhead of the deployed protections.

The former approach is not impeccable either. Offloading security features (e.g., code integrity [98], XOM [52], data integrity [116]) to commodity VMMs leads to a flat increase of the virtualization overhead (i.e., "blanket approach;" no targeted or agile hardening), and an even larger TCB, which, in turn, necessitates the deployment of hypervisor protection mechanisms [117], some of which are implemented in super-privileged CPU modes [9]. Considering the above, and the fact that hypervisor exploits are becoming an indispensable part of the attackers' arsenal [48], we investigate a previously unexplored point in the design space continuum. More specifically, our proposed self-protection approach to R^X enforcement: (a) does not require VMMs [52] or software executing in super-privileged CPU modes [9]; (b) avoids (nesting) virtualization overheads; and (c) is in par with recent industry efforts [25]. Lastly, kR^X enables R^X capabilities even in systems that lack virtualization support.

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

Note that, 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 [32]) is geared towards the kernel setting. CPH was designed with support for C++, dynamic linking, and just-in-time (JIT) compilation in mind. In contrast, commodity OSes: (a) do not support C++ in kernel mode, hence `vtable` and exception handling, and COOP [101] 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; (c) have limited support for JIT code in kernel space (typically to facilitate tracing and packet filtering [29]). These reasons prompted us to study 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 software fault isolation (SFI) [83, 103, 115, 123]. 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 [83], NaCl [103, 123]) are designed for confining the control flow and memory write operations of the sandboxed code, typically by imposing a canonical layout [103], bit-masking memory writes [115], and instrumenting computed branch instructions [83]. 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 [123]. In contrast, kR^X focuses on the *read* operations of benign code that can be abused to disclose memory [68]. Memory reads are usually ignored by conventional SFI schemes, due to the non-trivial overhead associated with their instrumentation [19, 83]. 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 pure software-only R^X scheme; (b) `kR^X-MPX`: a hardware-assisted R^X scheme, which exploits the Intel Memory Protection Extensions (MPX) [60] to (almost) eliminate the protection overhead; and (c) `kR^X-KAS`: a new kernel space layout that facilitates the efficient R^X enforcement by (a) and (b).

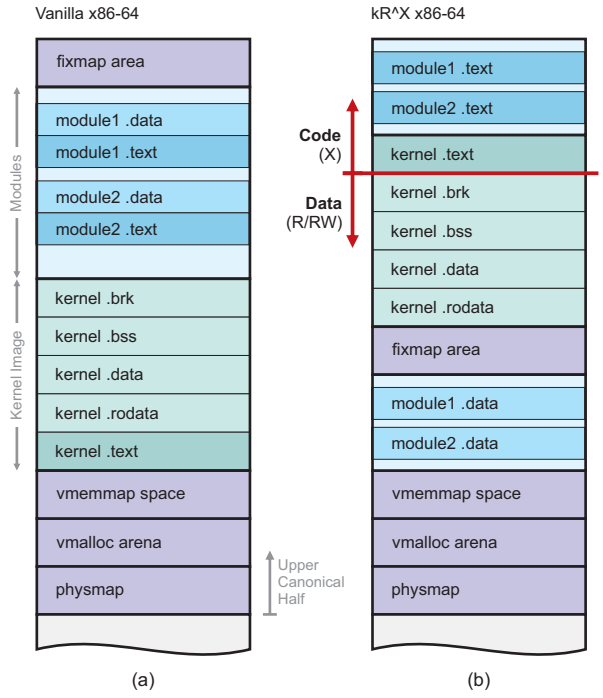

Figure 1. The Linux kernel space layout in x86-64: (a) vanilla and (b) `kR^X-KAS`. The kernel image and `modules` regions may contain additional (ELF) sections; only the standard ones are shown.

5.1.1 `kR^X-KAS`

The x86-64 architecture uses 48-bit virtual addresses that are sign-extended to 64 bits (bits `[48:63]` are copies of bit `[47]`), splitting the 64-bit virtual address space in two halves of 128TB each. In Linux, kernel space occupies the upper canonical half (`[0xFFFF800000000000:`$2^{64}-1$`]`), and is further divided into six regions (see Figure 1(a)) [66]: `fixmap`, `modules`, kernel image, `vmemmap` space, `vmalloc` arena, and `physmap`.

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`), kR^X relies on a modified kernel layout that maps code and data into *disjoint*, contiguous regions (see Figure 1(b)). 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`, which is "pushed" towards lower addresses, and `modules`, which is replaced by two newly-created areas: `modules_text` and `modules_data`. `modules_text` occupies the original `modules` area, whereas `modules_data` is placed right below `fixmap`. The size of both regions is configurable, with the default value set to 512MB.[4]

[4] The default setting was selected by dividing the original `modules` area in two equally-sized parts (i.e., `sizeof(modules)/2`).

Kernel Image The kernel image is loaded in its assigned location by a staged bootstrap process. Conventionally, the `.text` section is placed at the beginning of the image, followed by standard (i.e., `.rodata`, `.data`, `.bss`, `.brk`) and kernel-specific sections [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 up in the data region.[5] 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`, while the rest of the (allocatable) sections of the ELF object are loaded in `modules_data`. Once everything is copied in kernel space, relocation and symbol binding take place (eager loading [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 [63]. Hence, as physical memory is alloted to the kernel image, and modules, the existence of `physmap` results in *address aliasing*; virtual address aliases, or *synonyms* [67], 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 from `physmap` any synonym pages of `.text` sections (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 [106]).

Alternative Layouts `kR^X-KAS` has several advantages over the address space layouts imposed by SFI-based schemes (e.g., NaCl [123], 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. Second, the use of bit-masking confinement (similarly to NaCl [123] and LR^2 [19]), in the kernel setting, requires a radically different set of memory allocators to cope with the alignment constrains of bit-masking. In contrast, the layout of `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, like KASLR [40], are readily supported without requiring any kernel code change or redesign.

[5] 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.

Finally, in x86-64, the *code model* (`-mcmodel=kernel`) used generates code for the negative 2GB of the address space [46]. This model requires the `.text` section of the kernel image and modules, and their respective global data sections, to be not more than 2GB apart. The reason is that the offset of the x86-64 `%rip`-relative `mov` instructions is only 32 bits. `kR^X-KAS` respects this constraint, whereas a scheme like LR^2 (halved address space) would require transitioning to `-mcmodel=large`, which incurs additional overhead, as it rules out `%rip`-relative addressing. Interestingly, the development of `kR^X-KAS` helped uncover two kernel bugs (one security related)—Appendix A provides more details.

5.1.2 `kR^X-SFI`

`kR^X-SFI` is a software-only R^X scheme that targets modern (x86-64) platforms. Once the `kR^X-KAS` layout is in place, R^X can be enforced by checking *all* memory reads and making sure they fall within the data region (addresses below `_krx_edata`). As bit-masking load instructions is not an option, due to the non-canonical layout, `kR^X-SFI` employs *range checks* (RCs) instead. The range checks are placed (at compile time) right before memory read operations, ensuring (at runtime) that the *effective addresses* of reads are legitimate. We will be using the example code of Figure 2 to present the internals of `kR^X-SFI`. The original code excerpt is listed in Figure 2(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) [76]. 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 [83, 103, 115], but earlier work focuses on RISC-based architectures [19, 115] or fine tunes bit-masking confinement [83]. 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 2(a). First, the effective address of the memory read is loaded by `lea` in the `%r11` scratch register, and is subsequently checked against the end of the data region (`cmp`). If the effective address falls above `_krx_edata` (`ja`), then this is a R^X violation, as the read tries to access the code region. In this case, `krx_handler()` is invoked (`callq`) to handle the violation; our default handler appends a warning message to the kernel log and halts the system. 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 the range check (`cmp`).

pushfq/popfq Elimination (`O1`) Spilling and filling the `%rflags` register is expensive [81]. However, we can eliminate *redundant* `pushfq-popfq` pairs by performing a liveness analysis on `%rflags`. Figure 2(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 2(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 2(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 ($\texttt{offset}_a$ `!=` $\texttt{offset}_b$), 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; (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 2(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, effectively confining all three memory reads. If `%rsi + 0x154 < _krx_edata`, then `%rsi + 0x140` and `%rsi + 0x130`

[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 over-preserve).

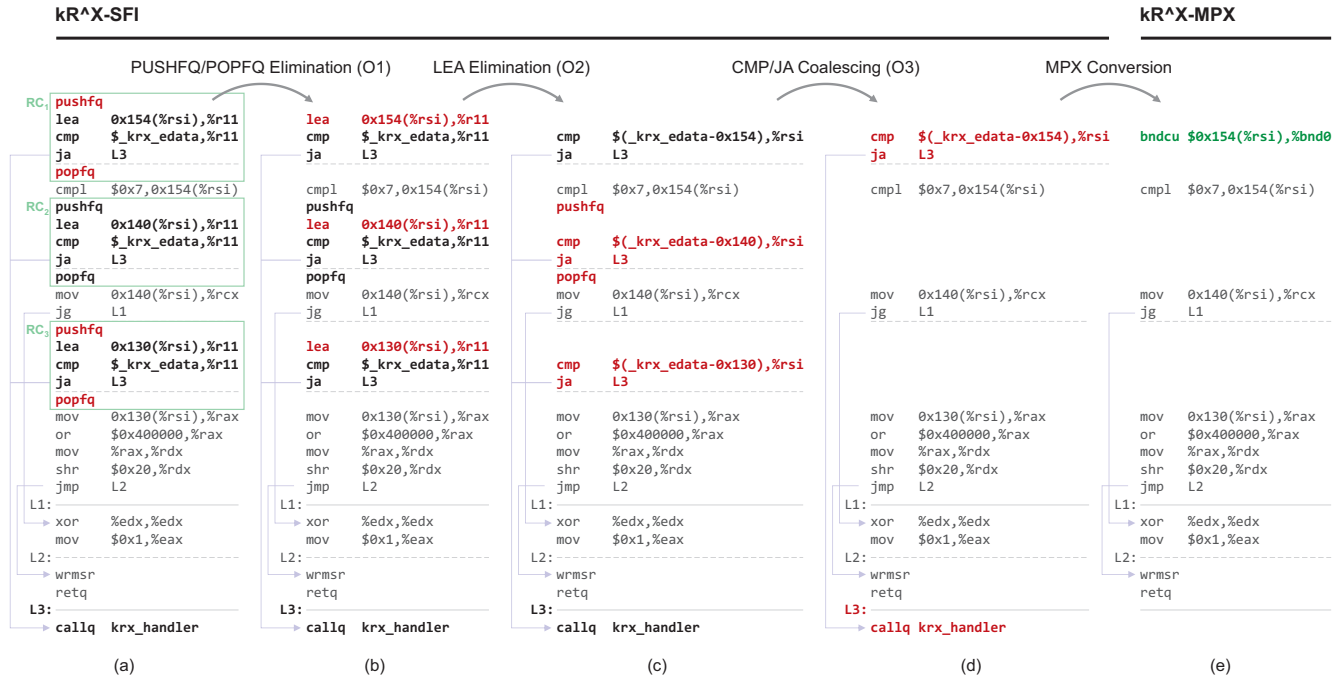

Figure 2. The different optimization phases of `kR^X-SFI` (a)–(d) and `kR^X-MPX` (e).

are guaranteed to "point" below `_krx_edata`, as long as `%rsi` does not change between the RC and the respective memory reads. The reason we require `%rsi` not to be spilled is to prevent temporal attacks, like those demonstrated by Conti et al. [24]. About one out of every two RCs can be eliminated using RC coalescing.

String Operations The x86 string operations [59], namely `cmps`, `lods`, `movs`, and `scas`, read memory via the `%rsi` register (except `scas`, which uses `%rdi`). `kR^X-SFI` instruments these instructions with RCs that check (`%rsi`) or (`%rdi`), accordingly. If the string operation is `rep`-prefixed, the RC is placed *after* the confined instruction, checking `%rsi` (or `%rdi`) once the respective operation is complete.[7]

Stack Reads If the stack pointer (`%rsp`) is used with a scaled index register [59], the read is instrumented with a range check as usual. However, if the effective address of a stack read consists only of `(%rsp)` or `offset(%rsp)`, the range check can be eliminated by spacing appropriately the code and data regions. Recall, though, that attackers may *pivot* `%rsp` anywhere inside the data region. By repeatedly positioning `%rsp` at (or close to) `_krx_edata`, they could take advantage of uninstrumented stack reads and leak up to `offset` bytes from the code region (assuming they control the contents at, or close to, `_krx_edata` for reconciling the effects of the dislocated stack pointer). `kR^X-SFI` deals with this slim possibility by placing a guard section (namely `.krx_phantom`), between `_krx_edata` and the beginning of the code region. Its size is set to be greater than the maximum `offset` of all `%rsp`-based memory reads.

Safe Reads Absolute and `%rip`-relative memory reads are not instrumented with range checks, as their effective addresses are encoded within the instruction itself and cannot be modified at runtime due to W^X. Safe reads account for 4% of all memory reads.

5.1.3 `kR^X-MPX`

`kR^X-MPX` is a hardware-assisted, R^X scheme that takes advantage of the MPX (Memory Protection Extensions) [60] feature, 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.[8]

MPX introduces four new bounds registers (`%bnd0`–`%bnd3`), each consisting of two 64-bit parts (`lb`; lower bound, `ub`; upper bound). `kR^X-MPX` uses `%bnd0` to implement RCs and initializes it as follows: `lb = 0x0` and `ub = _krx_edata`, effectively covering everything up to the end of the data region. Memory reads are prefixed with a RC as before (at compile time), but the [`lea`, `cmp`, `ja`] triplet is now replaced with a *single* MPX instruction (`bndcu`), which checks the effective address of the read against the upper bound of `%bnd0`.

[7] We always 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] Interestingly, although the Linux kernel already includes the necessary infrastructure to provide MPX support in user programs, kernel developers are reluctant to use MPX within the kernel itself [30].

Figure 2(e) illustrates the instrumentation performed by kR^X-MPX. Note that bndcu does not alter %rflags, so there is no need to preserve it. Also, the checked effective address is encoded in the MPX instruction itself, rendering the use of lea with a scratch register unnecessary, while violations trigger a CPU exception (#BR), obviating the need to invoke krx_handler() explicitly. In a nutshell, optimizations O1 and O2 are not relevant when MPX is used to implement range checks, whereas O3 (RC coalescing) is used as before. 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.

5.2 Fine-grained KASLR

With kR^X-{SFI, MPX} 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 [24, 37]. The use of code perturbation or randomization to hinder code-reuse attacks has been studied extensively in the past [13, 38, 53, 57, 64, 87, 118]. Previous research, however, either did not consider resilience to indirect JIT-ROP [24, 37], or focused on schemes geared towards user-land code [19, 32]. 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 [38, 118], whereas at the section (.text) level, we perform function permutation [13, 64].

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 [38]. Yet, as the achieved randomness depends on B, routines with a few basic blocks end up having extremely low randomization entropy. For instance, $\sim$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 [38, 118], and we augmented kR^X to resolve it as follows.

Starting with k, the number of randomization entropy bits *per function* we seek to achieve (a compile-time parameter), we first slice routines at call sites (i.e., code blocks ending with callq). 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 Note that, despite code block permutation, an attacker that discloses a function pointer would still be able to reuse gadgets from the entry code block of the respective function. To prevent this, functions always begin with a phantom block: the first instruction of each function is a jmp instruction that transfers control to the original first code block. Hence, an attacker armed with a leaked function pointer can only reuse a whole function, which is not a viable strategy, as we further discuss in Section 7.3.

5.2.2 Return Address Protection

Return addresses are stored in kernel stacks, which, in turn, are allocated from the readable data (physmap) region of kR^X-KAS [63]. Conti et al. demonstrated an indirect JIT-ROP attack that relies on harvesting return addresses from stacks [24]. 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, 91, 120]. 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 summary, unmangled return addresses are pushed into the kernel stack by the caller (callq), get encrypted by the callee, and remain encrypted until the callee returns (retq) 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. Finally, all xkey variables are merged into a contiguous region at link time, and replenished with random values at boot time (kernel image) or load time (modules).

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 Sec. 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)

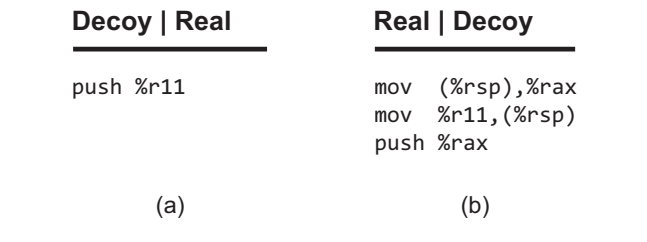

Figure 3. Instrumentation code (function prologue) to place the decoy return address (a) below or (b) above the real one.

instructions, whose execution raises an exception [31]. For instance, `mov $0xcc,%r11` is a phantom instruction; apart from changing the value of `%r11`, it does not alter the CPU or memory state. The opcodes of the instruction are the following: `49 C7 C3 CC 00 00 00`. 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. Armed with that information, the callee either: (a) places the address of the tripwire right below the saved return address on the stack; or (b) relocates the return address so that the address of the tripwire is stored where the return address used to be, followed by the saved return address (Figure 3). 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`: the address of the tripwire).[9] The exact ordering is decided randomly at compile time.

kR^X always slices routines at call sites. Therefore, by randomly inserting phantom instructions in routine code, their relative placement to return sites cannot be determined in advance (code block randomization perturbs them independently). As a result, although return address-decoy pairs can be harvested from the kernel stack(s), the attacker cannot differentiate which is which, because that information is encoded in each routine's code, which is not readable (R^X).

The net result is that `call`-preceded gadgets [20, 36, 54] are coupled with a *pair* of return addresses (`R` and `D`), thereby forcing the attacker to randomly choose one of them. If `n` `call`-preceded gadgets are required for an indirect JIT-ROP attack, the attacker will succeed (i.e., correctly guess the real return address in all cases) with a probability $\mathtt{P_{succ}} = 1/2^{\mathtt{n}}$. Wrong guesses raise CPU exceptions (e.g., `#BR`; `int 3`).

[9] Stack offsets are adjusted whenever necessary: if frame pointers are used, negative `%rbp` offsets are decreased by `sizeof(unsigned long)`; if frame pointers are omitted, `%rsp`-based accesses to non-local variables are increased by `sizeof(unsigned long)`. Function epilogues are emitted so that, depending on the scheme employed, they make use of the real return address (i.e., by adjusting `%rsp` before `retq` and tail calls).

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 does leave a window open for an attacker to probe the stack and leak unencrypted/real return addresses [24]. In order for an attacker 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. Evidently, winning such a race, in a *reliable* way, is not easily attainable. Nevertheless, 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: $[\mathrm{RS}_1 \hat{}\, \mathtt{xkey_f}]$. Likewise, if `f` is invoked from CS_2, RS_2 will be saved as $[\mathrm{RS}_2 \hat{}\, \mathtt{xkey_f}]$. Hence, if an attacker manages to leak both "ciphertexts," though they cannot recover RS_1, RS_2, or $\mathtt{xkey_f}$, they may replace $[\mathrm{RS}_1 \hat{}\, \mathtt{xkey_f}]$ with $[\mathrm{RS}_2 \hat{}\, \mathtt{xkey_f}]$ (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).[10]

Substitution attacks resemble the techniques for overcoming coarse-grained CFI by stitching together `call`-preceded gadgets [20, 36, 54]. However, in such CFI bypasses, *any* `call`-preceded gadget can be used as part of a code-reuse payload, whereas in a substitution attack, for every function `f`, the (hijacked) control flow can only be redirected to the *valid* return sites of `f`, and, in particular, to the subset of those valid sites that can be leaked *dynamically* (i.e., at runtime). Leaving aside the fact that the number of `call`-preceded gadgets, at the attacker's disposal, is highly limited in such scenarios, both our return address protection schemes aim at thwarting (indirect) JIT-ROP, and, therefore, are not geared towards ensuring the integrity of code pointers [70]. In any case, they can be easily complemented with a register randomization scheme [32, 87], which foils `call`-preceded gadget chaining [19].

[10] Replacing $[\mathrm{RS}_1 \hat{}\, \mathtt{xkey_f}]$, or $[\mathrm{RS}_2 \hat{}\, \mathtt{xkey_f}]$, with *any* harvested (and therefore encrypted) return address, say $[\mathrm{RS}_n \hat{}\, \mathtt{xkey_f}]$, is not a viable strategy because the respective return sites (RS_1/RS_2, RS_n) are encrypted with different keys ($\mathtt{xkey_f}$, $\mathtt{xkey_f}$,)—under return address encryption (`X`), substitution attacks are only possible among return addresses encrypted with the same `xkey`.

6. Implementation

Toolchain We implemented kR^X-SFI and kR^X-MPX 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 (Sections 5.1.2 and 5.1.3); and (b) randomize code blocks and protect return addresses (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 vartrack RTL optimization pass, by calling GCC's register_callback() function and hooking with the pass manager [62]. The reasons for choosing to implement our instrumentation logic at the RTL level, and not as annotations to the GENERIC or GIMPLE IR, are the following. First, by applying our instrumentation after the important optimizations have been performed, which may result into instructions being moved or transformed, it is guaranteed that only relevant code will be protected. Second, any implicit memory reads that are exposed later in the translation process are not neglected. Third, the inserted range checks are tightly coupled with the corresponding unsafe memory reads. This way, the checks are protected from being removed or shifted away from the respective read operations, due to subsequent optimization passes [24]. 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 (Section 5.1.1) is 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); (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) reserve %bnd0 and load it with the value of _krx_edata (MPX only); (f) place .text section(s) at the end of the vmlinux image and permute their functions (vmlinux.lds.S); (g) map the kernel image in kR^X-KAS, so that executable code resides in the non-readable region.

Note that although kR^X requires patching the OS kernel, and (re)compiling with custom GCC plugins, it does support *mixed* code: i.e., both protected and unprotected modules; this design not only allows for incremental deployment and adoption, but also facilitates selective hardening [49].

Assembly Code Both krx and kaslr are implemented as RTL IR optimization passes, and, therefore, cannot handle *assembly* code (both "inline" or external). However, this is not a fundamental limitation of kR^X, but rather an implementation decision. In principle, the techniques presented in Section 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. (In fact, we have started implementing them in GNU as for achieving 100% code coverage.)

Legitimate Code Reads Kernel tracing and debugging (sub)systems, such as ftrace and KProbes [29], 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. 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.

7. Evaluation

We studied the runtime overhead of kR^X-{SFI, MPX}, both as standalone implementations, as well as 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 [84] for micro-benchmarking, and employed the Phoronix Test Suite (PTS) [97] to measure the performance impact on real-world applications. (Note that PTS is used by the Linux kernel developers to track performance regressions.) The reported LMBench and PTS 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 [84], focusing on two metrics: *latency* and *bandwidth* overhead. Specifically, we measured the additional latency imposed on: (a) critical system calls, like `open()/close()`, `read()/write()`, `select()`, `fstat()`, `mmap()/munmap()`; (b) mode switches (i.e., user mode to kernel mode and back) using the `null` system call; (c) process creation (`fork()+exit()`, `fork()+execve()`, `fork()+/bin/sh`); (d) signal installation (via `sigaction()`) and delivery; (e) protection faults and page faults; (f) pipe I/O and socket I/O (`AF_UNIX` and `AF_INET` TCP/UDP sockets). Moreover, we measured the bandwidth degradation on pipe, socket (`AF_UNIX` and `AF_INET` TCP), and file I/O.

Table 1 summarizes our results. The columns `SFI(-O0)`, `SFI(-O1)`, `SFI(-O2)`, `SFI(-O3)`, and `MPX` correspond to the overhead of RC-based (R^X) confinement alone. In addition, `SFI(-O0)`–`SFI(-O3)` illustrate the effect of `pushfq/popfq` elimination, `lea` elimination, and `cmp/ja` coalescing, when applied on an aggregate manner. The columns `D` and `X` correspond to the overhead of return address protection (`D`: return address decoys, `X`: return address encryption) *coupled* with fine-grained KASLR, whereas the last four columns (`SFI+D`, `SFI+X`, `MPX+D`, `MPX+X`) report the overhead of the full protection schemes that kR^X provides.

The software-only `kR^X-SFI` scheme incurs an overhead of up to 24.82% (avg. 10.86%) on latency and 6.43% (avg. 2.78%) on bandwidth. However, with hardware support (`kR^X-MPX`) the respective overheads decrease dramatically; latency: $\leq$ 6.27% (avg. 1.35%), bandwidth: $\leq$ 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.

In a nutshell, the impact of kR^X on I/O bandwidth ranges from negligible to moderate. As far as the latency is concerned, different kernel subsystems and services are affected dissimilarly; `open()/close()`, `read()/write()`, `fork()+execve()`, 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 [97]; PTS offers a number of *system* tests, such as `ApacheBench`, `DBench`, and `IOzone`, along with real-world workloads, like extracting and building the Linux kernel. Table 2 presents the overhead for each benchmark, under the different memory protection (`SFI`, `MPX`) and code diversification (`D`, `X`) schemes that kR^X provides.

On CPUs that lack 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%–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).

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] by Kemerlis et al. [63], 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, none of which remained correct.

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

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.[11] 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.

[11] 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.

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

Table 1. kR^X runtime overhead (% over vanilla Linux) on the LMBench micro-benchmark.

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 2. kR^X runtime overhead (% over vanilla Linux) on the Phoronix Test Suite.

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. Due to code block randomization, the knowledge of a return site cannot be used to infer the addresses of gadgets relative to the return site itself (the instructions following a return site are always placed in a permuted code block). Yet, an attacker can still leverage a return site to construct ROP payloads with `call`-preceded gadgets [20, 36, 54]. In kR^X, return addresses are either encrypted, and hence their leakage cannot convey any information regarding the placement of return sites, or "hidden" among decoy addresses, forcing the attacker to guess between two gadgets (i.e., the real one and the tripwire) for every `call`-preceded gadget used; if the payload consists of `n` such gadgets the probability of succeeding is $\mathtt{P_{succ}} = 1/2^{\mathtt{n}}$.

Regarding function pointers (i.e., addresses of function entry points that can be harvested from the stack, heap, or global data regions, including the interrupt vector table and system call table) or *leaked* return addresses (Section 5.3), due to function permutation, their leakage does not reveal anything about the immediate surrounding area of the disclosed routine. In addition, due to code block permutation, knowing any address of a function (i.e., 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 30 bits, the safest strategy for an attacker is to reuse whole functions. However, in x86-64 Linux kernels, function arguments are passed in registers [82], and that necessitates (in the general case) the use of gadgets for loading registers with the proper values. In essence, kR^X effectively restricts the attacker to data-only type of attacks on function pointers [107] (i.e., overwriting function pointers with the addresses of functions of the same, or lower, arity [42]).

8. Related Work

We already covered related work in the broader areas of code diversification and XOM in Section 1 and 2. Most of these efforts are geared towards userland applications, and, 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 [32, 33] (i.e., non-self-protection), secrecy of segment descriptors [10, 78] and custom page fault handling [11, 51] (i.e., non-realistic assumptions for the kernel setting), as well as designs that treat the kernel as part of the TCB. Besides, some of the proposed schemes suffer from high overheads, which are prohibitive for the OS kernel [37].

LR^2 [19] and KHide [52] 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 constrains) 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 (x86-64 vs. 32-bit ARM, kernel vs. user space); (b) can leverage hardware support when available (MPX); 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 code reuse attacks, but relies on a commodity VMM (KVM) to do so; kR^X adopts a self-protection-based approach instead. More importantly, KHide, in antithesis to kR^X, does not conceal return addresses, which is important for defending against indirect JIT-ROP attacks [24].

Live Re-randomization Giuffrida et al. [53] 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 (e.g., Linux, BSDs), while it incurs a significant runtime overhead for short re-randomization intervals. TASR [14] 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 [120] re-randomizes userland applications continuously, on the order of milliseconds, but treats the OS kernel as part of its TCB. Lastly, RuntimeASLR [79] re-randomizes the address space of server worker processes to prevent clone-probing attacks; such attacks are not applicable to kernel settings.

Other Kernel Defenses KCoFI [34] augments FreeBSD with support for coarse-grained CFI, whereas the system of Ge et al. [47] further rectifies the enforcement approach of HyperSafe [117] to implement a fine-grained CFI scheme for the kernels of MINIX and FreeBSD. In the same vein, PaX's RAP [91] 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, 36, 42, 54]. Heisenbyte [109] and NEAR [119] employ (the interesting concept of) destructive code reads to thwart attacks that rely on code disclosures (e.g., JIT-ROP). Alas, Snow et al. demonstrated that destructive code reads can be undermined with code inference attacks [106].

Li et al. [74] 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 [23, 54]. kR^X, on the other hand, provides comprehensive protection against all types of (known) code-reuse attacks. Finally, Song et al. proposed KENALI [107] to defend against data-only attacks. KENALI enforces kernel data flow integrity [22] 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).

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 paper, we investigated a previously unexplored point in the design space by presenting kR^X: a practical hardening scheme that fulfils the current lack of self-protection-based execute-only kernel memory. Implemented as a GCC plugin, kR^X is readily applicable on x86-64 Linux, it does not rely on a hypervisor or any other more privileged subsystem, it does not require any modifications to existing applications, and it incurs a low runtime overhead, benefiting from the availability of MPX support.

Availability

The prototype implementation of kR^X is available at: `http://nsl.cs.columbia.edu/projects/krx`

Acknowledgments

We thank the anonymous reviewers, our shepherd Haibo Chen, and Junfeng Yang for their valuable comments. This work was supported in part by the National Science Foundation (NSF) through grant CNS-13-18415, and the Office of Naval Research (ONR) through contract N00014-15-1-2378, 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, or Qualcomm.

References

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

[2] Analysis of jailbreakme v3 font exploit. `https://goo.gl/RGsgzc`, July 2011.

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

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

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

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

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proc. of ACM CCS*, pages 340–353, 2005.

[8] Andrea Bittau. Linux Kernel < 3.8.9 (x86_64) 'perf_swevent_init' Privilege Escalation. `https://www.exploit-db.com/exploits/26131/`, June 2013.

[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *Proc. of ACM CCS*, pages 90–102, 2014.

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

[11] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proc. of ACM CCS*, pages 1342–1353, 2014.

[12] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proc. of USENIX OSDI*, pages 423–436, 2010.

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

[14] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proc. of ACM CCS*, pages 268–279, 2015.

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

[16] D. P. Bovet. Special sections in Linux binaries. `https://lwn.net/Articles/531148/`, January 2013.

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

[18] Brad Spengler and Sorbo. Linux `perf_swevent_init` Privilege Escalation. `https://goo.gl/eLgE48`, March 2014.

[19] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proc. of NDSS*, 2016.

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

[21] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proc. of USENIX Sec*, pages 161–176, 2015.

[22] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proc. of USENIX OSDI*, pages 147–160, 2006.

[23] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proc. of ACM CCS*, pages 559–572, 2010.

[24] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks. In *Proc. of ACM CCS*, pages 952–963, 2015.

[25] K. Cook. Kernel Self Protection Project. `https://goo.gl/KsN0t8`.

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

[27] J. Corbet. An updated guide to `debugfs`. `https://lwn.net/Articles/334546/`, May 2009.

[28] J. Corbet. Supervisor mode access prevention. `https://lwn.net/Articles/517475/`, October 2012.

[29] J. Corbet. BPF: the universal in-kernel virtual machine. `https://lwn.net/Articles/599755/`, May 2014.

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

[31] S. Crane, P. Larsen, S. Brunthaler, and M. Franz. Booby Trapping Software. In *Proc. of NSPW*, pages 95–106, 2013.

[32] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proc. of IEEE S&P*, pages 763–780, 2015.

[33] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table Randomization and Protection Against Function-Reuse Attacks. In *Proc. of ACM CCS*, pages 243–255, 2015.

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

[35] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve. Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation. In *Proc. of ACM ASPLOS*, pages 191–206, 2015.

[36] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proc. of USENIX Sec*, pages 401–416, 2014.

[37] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proc. of NDSS*, 2015.

[38] L. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi. Gadge Me if You Can: Secure and Efficient Ad-hoc Instruction-level Randomization for x86 and ARM. In *Proc. of ACM ASIACCS*, pages 299–310, 2013.

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

[40] J. Edge. Kernel address space layout randomization. https://lwn.net/Articles/569635/, October 2013.

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

[42] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proc. of ACM CCS*, pages 901–913, 2015.

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

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

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

[46] GCC online documentation. Intel 386 and AMD x86-64 Options. https://goo.gl/38gK86.

[47] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proc. of IEEE EuroS&P*, 2016.

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

[49] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis. Adaptive Defenses for Commodity Software through Virtual Application Partitioning. In *Proc. of CCS*, pages 133–144, 2012.

[50] M. Gillespie. Best Practices for Paravirtualization Enhancements from Intel® Virtualization Technology: EPT and VT-d. https://goo.gl/LLlAZK, January 2015.

[51] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proc. of ACM CODASPY*, pages 325–336, 2015.

[52] J. Gionta, W. Enck, and P. Larsen. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *Proc. of IEEE CNS*, 2016.

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

[54] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.

[55] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proc. of USENIX Sec*, pages 417–432, 2014.

[56] D. Hansen. [RFC] x86: Memory protection keys. https://lwn.net/Articles/643617/, May 2015.

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

[58] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proc. of USENIX Sec*, pages 384–398, 2009.

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

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

[61] Intel® OS Guard (SMEP). Intel® Xeon® Processor E5-2600 V2 Product Family Technical Overview. https://goo.gl/mS5Ile, October 2013.

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

[63] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *Proc. of USENIX Sec*, pages 957–972, 2014.

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

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

[66] A. Kleen. Memory Layout on amd64 Linux. https://goo.gl/BtvguP, July 2004.

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

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

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

[70] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proc. of USENIX OSDI*, pages 147–163, 2014.

[71] M. Larkin. Kernel W^X Improvements In OpenBSD. In *Hackfest*, 2015.

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

[73] J. Lee, H. Ham, I. Kim, and J. Song. POSTER: Page Table Manipulation Attack. In *Proc. of ACM CCS*, pages 1644–1646, 2015.

[74] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating Return-Oriented Rootkits With "*Return-less*" Kernels. In *Proc. of EuroSys*, pages 195–208, 2010.

[75] S. Liakh. NX protection for kernel data. https://lwn.net/Articles/342266/, July 2009.

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

[77] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proc. of ACM CCS*, pages 1607–1619, 2015.

[78] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proc. of ACM CCS*, pages 280–291, 2015.

[79] K. Lu, S. Nürnberger, M. Backes, and W. Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proc. of NDSS*, 2016.

[80] K. Lu, C. Song, T. Kim, and W. Lee. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *Proc. of ACM CCS*, pages 920–932, 2016.

[81] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of ACM PLDI*, pages 190–200, 2005.

[82] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface. `http://www.x86-64.org/documentation/abi.pdf`, October 2013.

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

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

[85] B. Niu and G. Tan. Modular Control-flow Integrity. In *Proc. of ACM PLDI*, pages 577–587, 2014.

[86] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *Proc. of ACM CCS*, pages 914–926, 2015.

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

[88] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. of USENIX Sec*, pages 447–462, 2013.

[89] PaX Team. UDEREF/amd64. `https://goo.gl/iPuOVZ`, April 2010.

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

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

[92] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proc. of DIMVA*, pages 144–164, 2015.

[93] E. Perla and M. Oldani. *A Guide To Kernel Exploitation: Attacking the Core*, chapter Stairway to Successful Kernel Exploitation, pages 47–99. Elsevier, 2010.

[94] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proc. of ACM CCS*, pages 103–115, 2007.

[95] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proc. of ACSAC*, pages 309–318, 2013.

[96] G. J. Popek and D. A. Farber. A Model for Verification of Data Security in Operating Systems. *Commun. ACM*, 21(9): 737–749, September 1978.

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

[98] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-based Memory Shadowing. In *Proc. of RAID*, pages 1–20, 2008.

[99] D. Rosenberg. `kptr_restrict` for hiding kernel pointers. `https://lwn.net/Articles/420403/`, December 2010.

[100] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proc. of ACSAC*, pages 448–459, 2016.

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

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

[103] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proc. of USENIX Sec*, pages 1–11, 2010.

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

[105] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proc. of IEEE S&P*, pages 574–588, 2013.

[106] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proc. of IEEE S&P*, pages 954–968, 2016.

[107] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proc. of NDSS*, 2016.

[108] B. Spengler. Enlightenment Linux Kernel Exploitation Framework. `https://goo.gl/hDymQg`, December 2014.

[109] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks Using Destructive Code Reads. In *Proc. of ACM CCS*, pages 256–267, 2015.

[110] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of ACM ASPLOS*, pages 168–177, 2000.

[111] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proc. of USENIX Sec*, pages 941–955, 2014.

[112] A. van de Ven. Debug option to write-protect rodata: the write protect logic and config option. `https://goo.gl/shDf0o`, November 2005.

[113] A. van de Ven. Add -fstack-protector support to the kernel. https://lwn.net/Articles/193307/, July 2006.

[114] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic Hooks: Hiding Control Flow Changes Within Non-control Data. In *Proc. of USENIX Sec*, pages 813–828, 2014.

[115] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient Software-based Fault Isolation. In *Proc. of ACM SOSP*, pages 203–216, 1993.

[116] X. Wang, Y. Chen, Z. Wang, Y. Qi, and Y. Zhou. SecPod: a Framework for Virtualization-based Security Systems. In *Proc. of USENIX ATC*, pages 347–360, 2015.

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

[118] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of ACM CCS*, pages 157–168, 2012.

[119] J. Werner, G. Baltas, R. Dallara, N. Otternes, K. Snow, F. Monrose, and M. Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proc. of ACM ASIACCS*.

[120] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proc. of USENIX OSDI*, pages 367–382, 2016.

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

[122] W. Xu and Y. Fu. Own Your Android! Yet Another Universal Root. In *Proc. of USENIX WOOT*, 2015.

[123] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proc. of IEEE S&P*, pages 79–93, 2009.

[124] F. Yu. Enable/Disable Supervisor Mode Execution Protection. https://goo.gl/utKHno, May 2011.

[125] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. of IEEE S&P*, pages 559–573, 2013.

[126] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proc. of USENIX Sec*, pages 337–352, 2013.

A. Discovered Kernel Bugs

During the development of kR^X-KAS, we discovered two kernel bugs. The first one, which is security critical, results in memory being accidentally marked as *executable*. In the x86 architecture, the MMU utilizes a multi-level page table hierarchy for mapping virtual to physical addresses. When the Physical Address Extension (PAE) [59] mode is enabled, which is the default nowadays as non-executable protection is only available under PAE mode, each page table entry is 64-bit wide, and except from addressing information also holds flags that define properties of the mapped page(s) (e.g., PRESENT, ACCESSED). Often, multiple adjacent pages sharing the same flags are coalesced to larger memory areas (e.g., 512 4KB pages can be combined to form a single 2MB page) to reduce TLB pollution [41].

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

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