Improved Architectures for Secure Intra-process Isolation

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Richard Joseph Connor

August 2021

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Traditionally, operating system security in practice has largely focused on inter-process isolation: limiting a process's access to shared system resources or resources owned by another process. In contrast, there are usually no restrictions on memory access between different components within the same process. An executable and set of libraries in the same process both have full access to each other's resources, which violates the principle of least privilege and can exacerbate the impact of security bugs. For example, an application that uses a cryptography library may never need to access encryption keys directly, yet a security vulnerability in the application (or any of its libraries) could still allow an attacker to access the encryption keys used by the cryptography library even if the library has no bugs of its own. This is not just theoretical; in 2017, a bug in an HTML parser leaked internal private keys used by a major CDN provider [8]. In a similar case, the infamous Heartbleed bug, an out-of-bounds memory access vulnerability in the packet parsing code of OpenSSL [11], saw widespread real-world exploitation [1]. This potential exposure necessitated expensive key rotations for thousands of websites [29]. In both cases, private keys were exposed even though the actual vulnerable code resided in subroutines that *never handled cryptographic keys*.

Intra-process isolation hopes to improve on this situation by enforcing finer-grained separation of resources. Various proposals have offered solutions for limiting access to designated memory regions to "components" such as particular threads, libraries, or even arbitrary snippets of code [77, 59, 53, 52, 46]. However, most of them suffer from one of

two problems: high overhead during execution, or high cost of switching between isolated components [72]. Recently, two concurrent works, called Hodor [38] and ERIM [72], set forth unique *hardware-assisted* intra-process memory isolation. These two systems achieve dramatically lower overhead by exploiting "memory protection keys for userspace", a new hardware feature available in some recent x86 processors [43]. Protection keys for userspace (often abbreviated as either PKU or MPK) incur no significant overhead during ordinary execution and only a small cost when switching between components [72].

While these hardware-assisted intra-process isolation systems seem to provide the "best of both worlds" by offering finer-grained access control within a single process at very low overhead, their design also presents a unique threat model and attack surface. This threat model uniquely assumes that the attacker is able to execute nearly arbitrary machine code in the untrusted component even though, from the kernel's perspective, the untrusted component has the same privileges as the trusted component. While the sandbox does monitor and restrict certain syscalls, this approach of retrofitting a new security boundary onto the kernel by syscall interception is error-prone and difficult to formally analyze. Syscalls that are known to break intra-process security boundaries are blocked on an ad-hoc basis, but there is no guarantee that no dangerous kernel interface is overlooked. Starting from this observation, we develop the following contributions:

- Demonstrate, classify, and analyze a variety of novel vulnerabilities affecting hardware-assisted intra-process isolation relying on static code invariants;
- Analyze the root cause of these vulnerabilities in terms of flawed assumptions, kernel threat model discrepancies, and practical challenges of secure implementation for security retrofitting;
- More closely examine the W^X assumption in the context of academic security literature, showing how its usual conception is not borne out by reality;
- Generalize one attack technique developed for intra-process isolation, demonstrating how it can bypass fully-precise control-flow integrity (CFI) with shadow stacks under realistic circumstances;

- Design and prototype the `xlock` system, which attempts to bring real-world systems into line with the W^X as it is commonly used, using a development model that emphasizes architecturally preventing whole classes of known attacks;

- Design and prototype a hybrid model (coprocesses) for intra-process isolation that continues to exploit userspace-only context switches for efficiency while also informing the kernel of intra-process security boundaries;

- Demonstrate that the coprocess prototype has an acceptable performance cost in a realistic application, protecting long-term private keys in memory for a web sever running SSL.

1.1 Evaluating the Security of PKU-based Sandboxes

In the first section of this work, we evaluate the security of proposed PKU-based sandboxes in a realistic context, and find that both systems are vulnerable to similar classes of software attacks. We group the attacks into a few families of issues: subverting memory access permissions; modifying code by rearranging mappings, controlling PKRU through the kernel, race conditions, interfering with non-memory process resources, and changing permissions directly. We detail several practical exploits that circumvent the isolation promised by the system and allow the untrusted component to access all protected memory. In many cases, an identical attack works against both systems, ERIM and Hodor. Using prototype code available for ERIM, we tested 10 proof-of-concept exploits (listed later in Table 3.1) and found that all of them succeeded in accessing protected memory from an untrusted component. We also expect eight of these attacks to succeed against Hodor with minimal changes.

Our attacks exploit flawed assumptions shared by both systems, such as accessing PKU-protected memory through the kernel, modifying in-process code that is presumed to be immutable, or manipulating the behavior of a trusted component in unexpected ways. These issues do not represent bugs in the Linux kernel or in PKU's design or hardware implementation. Instead, we argue that these attacks stem from a common root cause: **the**

threat model for secure in-process isolation is fundamentally at odds with the threat model of the PKU feature and the Linux kernel. PKU can be controlled with an unprivileged instruction and so is not designed to protect against malicious code that intends to elevate its own privileges. Meanwhile, the Linux kernel has a highly permissive process model which allows processes a great deal of control over their own resources and operation. Thus, Linux kernel developers have made no attempt at absolute enforcement of PKU permissions [37]. Consequently, we *discover and exploit a large attack surface of unprivileged syscalls affecting intra-process resources* which can interfere with the security guarantees of a PKU-based sandbox. For example, the Linux syscall `process_vm_readv` intentionally allows an unprivileged process to read its own memory without checking PKU permissions.

We discuss possible mitigations and measure their potential impact on performance. We show that ERIM's [72] `ptrace`-based sandbox, which implements the sandbox without kernel changes, incurs substantially higher performance penalties when we extend it to monitor syscalls that could otherwise be used to bypass the secure isolation. Some of the attacks detailed in Section 3.2 use only standard I/O syscalls, which are frequently used by legitimate applications and are thus expensive to monitor. Merely adding a check to monitor the `open` syscall decreases measured throughput by over 50% compared to ERIM's previously reported benchmarks on a web server [72]. This suggests that current **userspace-only** design for PKU-based intra-process isolation may require a steep performance penalty to operate securely without kernel changes. As a result, we conclude that secure solution to intra-process isolation requires at least some architectural changes.

1.2 Generalizing Memory Permissions Model Attacks

Our findings of flawed assumptions in hardware-assisted intra-process isolation systems raise another question: are these assumptions shared in other proposed systems or research prototypes? And if so, what are the security implications to the systems founded on the same assumptions? We discuss how the widely-used W^X assumption does not entirely hold

on Linux, and that this gap results in realistic attacks to bypass fine-grained control-flow integrity (CFI) systems.

1.2.1 The W^X Assumption

In particular, the W^X assumption is shared by many proposed exploit mitigation systems [14, 60, 32]. The assumption is usually formulated as a guarantee that code cannot be modified at runtime. Abadi et al's seminal 2005 article *Control-Flow Integrity* provides an example for how the assumption is typically posed [14]:

> **NWC** Non-Writable Code: It must not be possible for the program to modify code memory at runtime... NWC is already true on most current systems, except during the loading of dynamic libraries and runtime code-generation.

This assumption comprises two parts. First, it simply states process code cannot be modified at runtime. Second, it asserts that this assumption (at least usually) holds on modern operating systems. Other works in the literature use a similar assumption, either implicitly or stated explicitly [14, 60, 32].

Of course, this view is a slightly simplified model of memory permissions. Modern tool chains do generally produce applications without any writable and executable regions by default [2]. As long as the application does not make any changes to memory mappings, this property prevents any *userspace memory access* from directly modifying executable code. But, it is well-known that applications can mark new memory regions as executable at runtime. On Linux, the most obvious ways involve modifying page permissions using `mmap` or `mprotect`, but we also identify a variety of less obvious methods in Section 3.2.

Therefore, the W^X property as it exists on Linux is better understood as a narrow property governing individual userspace memory accesses, rather than a comprehensive policy guaranteeing code integrity. SELinux provides access controls that can deny *some* methods of modifying code at runtime [12, 68], but even these controls cannot guarantee total code integrity through the lifetime of a process (see Section 5.6 for details).

In contrast to the complexities of W^X in practice, references to the W^X assumption in the literature often treat it as a simple guarantee of code immutability. Thus, a gap

exists between W^X as it commonly implemented on real systems and as it is commonly used in the literature. This difference is not just theoretical. Many of our bypasses against PKU-based memory isolation exploit differences between the W^X model and reality.

In exploring the implications of this imperfect assumption, a natural next target is control-flow integrity (CFI). CFI systems rely on the W^X assumption [14, 60, 70, 80, 75], but they also use a stricter threat model compared to intra-process isolation. Intra-process isolation threat models presume that the attacker can launch a control-flow hijacking attack, while CFI models do not (since control-flow hijacking is exactly what CFI sets out to prevent). Therefore, extending our exploits to CFI systems is non-trivial as it requires us to work in a more constrained threat model.

1.2.2 Control-Flow Integrity

As W^X policies became more ubiquitous, attackers shifted towards code reuse attacks to sidestep these protections. In a code reuse attack, attackers hijack the program's control flow, transferring execution through a series of small snippets of *existing* program code (known as *gadgets*) to achieve a desired outcome. Each gadget consists of an existing chunk of code that contains some pertinent instructions (e.g. `pop` or `mov`) followed by an instruction that allows the attacker to direct control flow to the next link in the chain (e.g. `ret` or an indirect branch). In the absence of additional protections, typical programs and libraries contain many thousands of such gadgets, generally enabling Turing-complete computation by the attacker [66].

CFI is a potential mitigation for code reuse attacks [70, 80, 75, 78, 60, 14]. It protects so-called "forward-edges" (indirect branches) by instrumenting them with additional code that validates the branch target before taking the branch. *Fully-precise* static CFI would allow only branches that exist in the static control-flow graph of the program, but real-world CFI implementations are not fully-precise. Instead, practical CFI implementations approximate the possible control-flow graph of a program to varying degrees of precision [22]. Fully-precise CFI, then, serves as an idealized model for the "best possible" static CFI policy.

Shadow stacks [20, 28, 67] protect the "backward edge" (stored return addresses) from tampering, either by hiding them at an unpredictable address [48, 54] or restricting access

via hardware such as the MMU [20, 43]. Each function return accesses the shadow stack and uses it to validate the return pointer before branching to it.

When used together, CFI and shadow stacks attempt to constrain program execution, even in the face of an attacker with full read and write access to the program's address space. All branches, or forward edges, are limited by CFI to jump only to a target that is valid for that branch. All return instructions, or backward edges, are validated to only return to the correct call site.

1.2.3 The `proc/mem` Attack

One of the attacks developed against PKU-based intra-process isolation exploited access to the `proc/self/mem` pseudo-file in Linux's `procfs` filesystem, a file-like interface to a process's own virtual address space. We show how this method can be repurposed to bypass fully-precise CFI with shadow stacks. The attack depends on certain data-oriented gadgets, which find exist in both Nginx [10] and the GNU standard C library (glibc). We develop a proof-of-concept exploit using this technique against a simulated vulnerability in Nginx, and show how it can be used to inject arbitrary machine code into executable memory at runtime without violating any constraint of fine-grained CFI or shadow stacks.

1.3 Mitigations

We have identified substantial challenges in designing and implementing secure intra-process isolation, but the situation is not hopeless. We divide these challenges into two principal problems: securing static code invariants via runtime code integrity; and mitigating confused deputy attacks from the kernel. We address the former via a set of kernel patches designed to implement the W^X assumption, which then form the foundation for a modified intra-process isolation design addressing the latter.

1.3.1 W^X Violations

In Section 5, we design and implement a prototype framework for protecting runtime code integrity, called the `xlock` system. This system is intended to bridge the gaps between the W^X in theory and practice. `xlock` provides a simple interface in the form of a single system call that allows a process to "lock" its own code. We combine two techniques to systematically and comprehensively address the existing loopholes: first, we rule out an entire class of attacks by converting all executable memory to private, anonymous mapping; and second, we use static analysis to uncover kernel code paths that reach potential W^X violations and augment them with new access checks. We then test, measure, and discuss the performance impact of `xlock`.

1.3.2 Confused Deputies

In current PKU-based intra-process isolation designs, when a dangerous kernel interface is overlooked, this leads to a novel class of vulnerabilities that use the kernel as a "confused deputy" to break intra-process isolation. One intra-process component may cause the kernel to take actions on its behalf that allow it to access resources from another component. Crucially, these actions do not represent kernel bugs because they are *intended* behavior for the kernel, which does not recognize security boundaries within a process. Instead, the vulnerabilities arise from the incongruities in the kernel and sandbox security models.

Rather than try to scour millions of lines of kernel code for potential violations of the intra-process security model, we propose a fix that divides intra-process components along *existing* security boundaries. To this end, we design a system that allows the kernel to associate each intra-process component with a largely independent process control block, as if the component were a separate process. We call these virtual processes *coprocesses* because they exist alongside the process originating them, the *primary* process. Intra-process component switches remain entirely in userspace, and thus retain the associated performance benefits. When the process does make a context switch to the kernel (for a system call, interrupt, or similar event), the kernel accesses the PKRU register to identify the current component, and uses the associated coprocess for essentially all subsequent operations.

1.4 Outline

The remainder of this dissertation is divided into four main chapters. In Chapter 2, we provide the necessary background information for understanding the implementation details of existing intra-process isolation systems. The following chapter details several vulnerabilities common to ERIM and Hodor, two recent and efficient iterations on intra-process isolation that use PKU hardware to enforce memory isolation.

In Chapter 4 we expand on the imperfect W^X assumption that underly many of these vulnerabilities, and generalize one of the techniques to undermine this assumption to bypass control-flow integrity. Chapter 5 puts forward a possible mitigation, the `xlock` system, that reinforces the W^X assumption on Linux to prevent this attack.

Finally, Chapter 6 introduces a new hybrid model for intra-process isolation that combines efficient userspace context switches with kernel abstractions to provide strong isolation of non-memory resources. In combination with the `xlock` system, we argue that this design addresses known weaknesses in intra-process isolation in a way that is both efficient and reduces the likelihood of further implementation flaws.

Chapter 2

Background: Intra-process Isolation

Modern operating systems must provide a stable platform for a complex userspace application ecosystem. For this reason, the kernel carefully restricts interactions between processes. Process-specific resources - including register state and virtual address space - are opaque and inaccessible from other userspace processes. This inter-process *isolation* provides fault-tolerance by preventing application failures from cascading to unrelated processes and limits the scope of vulnerabilities and bugs. An attacker who compromises one running application ought not be able to leverage their position to peer into (or modify) another application's private memory space.

Isolation within an application could provide similar benefits at greater efficiency. Sensitive data in an application component (for example, the cryptographic library in a webserver) could be insulated from security vulnerabilities elsewhere in the application. However, simply placing application components in separate processes (or other context abstractions) can incur significant performance penalties (see [72] Section 6.5, and [38] Section 4.1). This performance cost is high because, despite their conceptual segregation, processes rely on the same underlying hardware for execution. *Switching* to a new active process generally requires that the kernel flush the transaction lookaside buffer (TLB) and restore the process' context, including its register state and virtual memory space. These considerations motivate more lightweight techniques for segmenting memory between components with varied levels of trust/access.

2.1 Related Work

Researchers have designed many systems providing some form of lightweight isolation within a process, with varying performance and security characteristics.

2.1.1 Kernel Abstractions

Some proposed systems propose novel kernel abstractions to facilitate low-cost switching between memory views, including shreds [25] and light-weight contexts (lwCs) [52]. Like threads, processes may own many lwCs, but these abstractions are not scheduling-related. Instead, each lwC carries its own (potentially overlapping) set of resources, including memory mappings. lwC switches are twice as fast as traditional context switches [52].

Wedge [17] introduces several new kernel concepts including *sthreads* (the application components), *tags* (permissions and memory objects), and *callgates* (predefined component entry points), and uses system calls for component switches. By default, an sthread cannot access any memory, file descriptor, system call, or call gate; instead, permissions for each of these resources must be individually granted by the programmer. These *default-deny* semantics prevent all attacks described in this work by default. However, our work does suggest that it may be difficult for a developer to predict which system calls may lead to isolation bypasses.

2.1.2 VMFUNC

Intel's VT-x virtualization extensions allow for unprivileged switching between extended page tables. MemSentry-VMFUNC [46], Hodor-VMFUNC [38], and SeCage [53] leverage this capability to present alternate memory views to trusted and untrusted application components. MemSentry-VMFUNC and SeCage require CFI to defend against an in-process adversary, while Hodor-VMFUNC uses dual-mapped trampolines for this purpose.

2.1.3 Static and Dynamic Bounds Checking

Type-safe programming languages [58] provide isolation via validity checks on memory accesses. These protections can prevent some bugs and security vulnerabilities, but they require the use of specific languages and do not apply to many existing software products. These languages are also sometimes unsuited to domains that require direct resource management by the programmer.

Software fault isolation (SFI) [74, 77, 23] can retrofit unsafe languages with similar checks, but at a significant performance penalty. Systems such as NativeClient [77] block *all* syscalls from the untrusted component and so are not vulnerable to any of the issues we describe in this paper. Read/write protection with MemSentry-SFI, a recent implementation, increased average runtime across activities in the SPEC CPU2006 benchmark suite by roughly 20% [46]. Hardware-supported checks (e.g. Intel MPX [43]) offer improved performance but still have significant runtime costs [64], and in some cases are vulnerable to Meltdown-based attacks [21]. SFI techniques also typically require a mechanism for control flow integrity (CFI) [14, 60, 75, 70, 80, 73, 79] to prevent in-process adversaries from simply bypassing bounds checks. CFI adds additional overhead, however; a recent MPX-backed technique introduced 9%-28% runtime overhead on SPEC CPU2006 activities [79]. A number of exploits in the literature challenge CFI system security [30, 33, 22].

2.1.4 Probabilistic Isolation

Probabilistic isolation techniques obscure a process' memory layout to hide sensitive regions like system libraries. A well-deployed example is address space layout randomization (ASLR) [69, 54]. While full ASLR can mitigate buffer overflow attacks, an entire family of effective side-channel bypasses [41, 36, 31, 35] casts doubt on the security of such approaches.

2.1.5 Trusted Execution

Trusted enclaves like ARM's TrustZone [15] and Intel's Software Guard Extensions (SGX) [42] provide yet another solution for cordoning sensitive software regions. The protections

afforded by these enclaves are robust (even kernel snooping is prohibited), but they are a heavyweight solution inappropriate for many applications [82], and can be vulnerable to side-channel attacks [34].

IMIX [32] and Microstache [59] are proposals to add instructions to the x86 ISA for isolating memory regions within a process. However, to defend against an in-process adversary, these extensions require CFI protection or code integrity.

2.2 Protection Keys for Userspace (PKU)

ERIM and Hodor use a relatively new hardware feature called Intel Protection Keys for Userspace (**PKU**) [43] to mediate memory access between in-process components. With PKU, memory access permissions can be changed directly in userspace in as little as 20 cycles. This overhead is an order of magnitude faster than the context switch required for a system call [72].

The PKU feature allows a process to control its own access to memory by tagging individual pages with a domain and governing access to each domain via a special register known as the Protection Key Rights for Userspace Pages, or PKRU[43]. The PKRU register can be written from userspace with the unprivileged `wrpkru` instruction.

This unprivileged access is a double-edged sword. On the one hand, it lets the process quickly modify memory access rules without invoking the kernel; on the other, it creates a problem for *secure* isolation. If an attacker exploits a vulnerability to gain control over one component, PKU's design does not prevent the attacker from executing code that writes to the PKRU register, allowing access to any domain. For this reason, PKU does not provide secure isolation on its own. An earlier system using PKU for in-process isolation, MemSentry-PKU [46], thus required additional protection such as CFI to prevent an attacker from simply changing their own memory access permissions.

Hodor and ERIM both address this problem by augmenting PKU with a software sandbox that aims to prevent components from making unauthorized changes to the PKRU register. We collectively refer to these systems as "PKU-based sandboxes". At a high level, both

systems detect `wrpkru` instructions in application and library code, and effectively neutralize all `wrpkru` instructions *except* ones that are immediately followed by either code that safely transfers control to a designated entry point of the trusted component or code that returns to the untrusted component (after validating the state of the PKRU register). The sandbox then monitors and restricts certain syscalls made by the process to prevent an untrusted component from introducing new executable code that violates these rules. We discuss the designs of ERIM and Hodor in more detail in Section 2.

2.2.1 Intel PKU

The PKU feature (available in Skylake or later Intel server processors since 2017 [43]) regulates memory accesses based on the state of a new 32-bit register, the **PKRU** register [43]. When PKU is enabled, each virtual memory page mapped by a process is associated with exactly one of 16 different regions or *protection keys*. Each key is associated with 2 bits in the PKRU register which controls access to reads and/or writes for that region. On each memory access, a hardware check compares the protection key of the accessed page with the state of the corresponding bits in the PKRU register in order to determine if the access is allowed. New `rdpkru` and `wrpkru` x86 instructions allow userspace reads and writes to PKRU. Because PKRU values are part of a processor core's extended register state, PKRU can also be written by the `xrstor` instruction, which is designed to restore register state after context switches. A number of recent PKU-based memory isolation frameworks have been proposed [46, 72, 38]. Here we focus on the most performant systems, i.e. those that do not require additional mechanisms to protect against code reuse attacks. In general, these systems require hardening the PKU feature by gatekeeping PKRU state.

2.2.2 ERIM

ERIM's security hinges on the absence of *unsafe* `wrpkru` instructions in executable pages of M_U (T is trusted not to call back into U or contain exploitable control flow vulnerabilities). Safe `wrpkru` instances are those immediately followed by either 1) a jump into T, or 2) a check

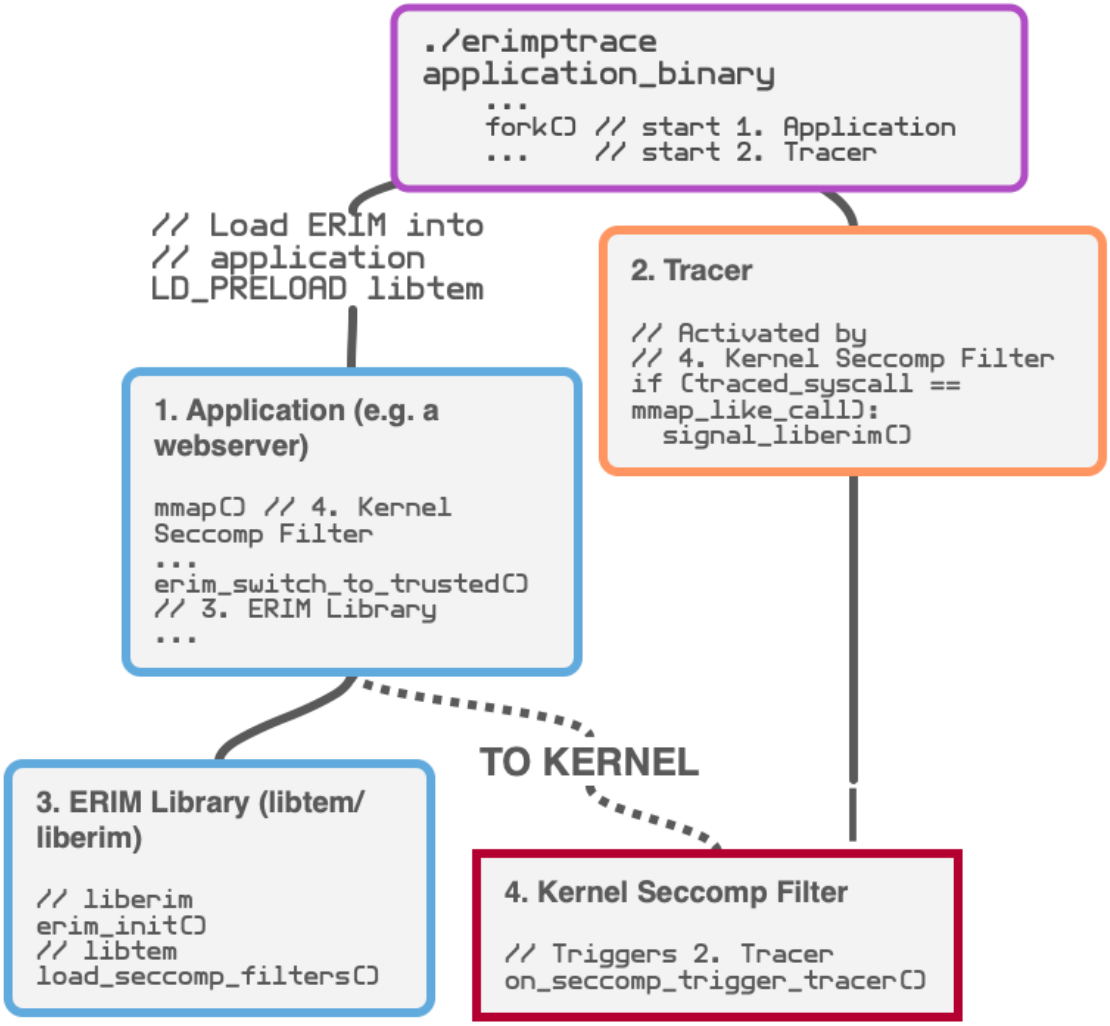

Figure 2.1: ERIM Architecture [72] with seccomp filters and process tracing.

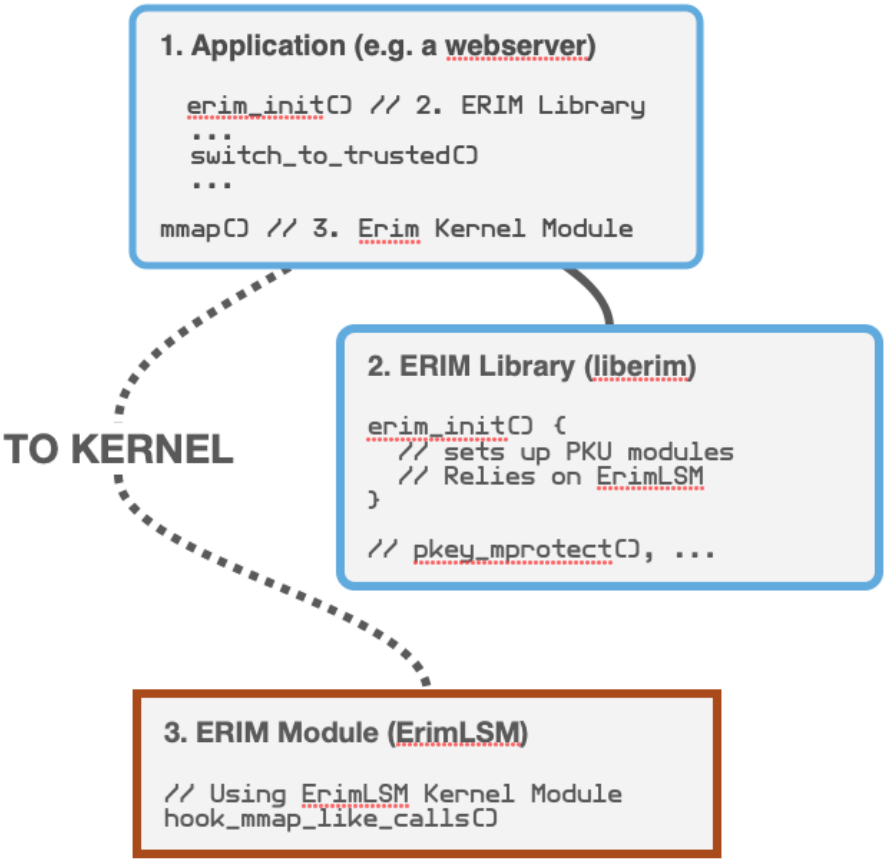

Figure 2.2: ERIM architecture [72] using a kernel module.

that M_T is locked by the PKRU register. Failing this check triggers process termination. An attacker who compromises U could exploit unsafe `wrpkru` instructions to unlock M_T from U; without such instructions, M_T is secure even without control flow integrity (CFI) in U. The call gates contain only safe `wrpkru`s by construction, but preventing intentional or unintentional occurrences of executable `wrpkru` instructions in M_U requires both *syscall interception* and *binary inspection*. Similar techniques are used to protect against unsafe `xrstor` instructions.

The purpose of binary inspection in ERIM is to scan memory for instructions that could be exploited to unlock M_T from U. When an ERIM-secured application is started, ERIM scans M_U for unsafe `wrpkru`/`xrstor` instructions. If any are detected, ERIM can be either replace them with safe variants (e.g., by adding a check that M_T remains locked) or immediately terminate the secured application. This initial inspection ensures no unsafe instructions are present in M_U at startup, but is insufficient on its own to prevent occurrences throughout the process' runtime - an attacker could map a new virtual memory page after the initial scan to introduce and execute unsafe instructions. ERIM intercepts related syscalls in an attempt to block this attack vector.

Interception can be performed via small kernel modifications (e.g. a Linux Security Module [62]), or by installing seccomp filters [4] that inform a tracer process. The seccomp filter with tracer mode is shown in Figure 2.1, while the kernel mode of operation is depicted in Figure 2.2. Figures 2.1 and 2.2 highlight how ERIM is deployed in practice, integrating with the host operating system and secured application binary.

In either case, `mmap` (mapping new pages), `mprotect` (page permission changes), and `pkey_mprotect` (page PKU region registration) syscalls from U that map executable memory pages are intercepted and redirected to ERIM functions in T. The memory is mapped only after the requested page sequence is scanned for `wrpkru` instructions within or across pages. Alternatively, ERIM can delay the scan and mark the sequence as "pending executable" for on-demand processing. Attempts to execute instructions from one of these pages will cause a fault handled by ERIM that signals a scan *for that page alone*. Scan failure results in termination of the program.

2.2.3 Hodor

As in ERIM, Hodor [38] separates trusted components T from untrusted components U, but different mechanisms are employed to create trust boundaries and prevent exploitation. More than two trust levels are supported; here we use T and U to simplify the description. In this section, we describe only the PKU-based variant (Hodor-PKU), but a more complete discussion in available in Section 2.1. Hodor always defines elements in T via library boundaries. A trusted loader ensures that the only entry points into T libraries exposed to U are gated by *trampolines* (analogues to ERIM call gates) that manage PKRU state. Like ERIM, Hodor deploys interception and inspection techniques to guard the PKRU before and during runtime.

The trusted loader is tasked with scanning for unsafe instructions that occur outside trampolines. Inspection is performed at startup, and again when any sequence of pages is marked executable by the protected application. Any pages containing unsafe instructions are marked pending executable. Calling into these marked pages will trigger a page fault, signaling the Hodor-modified kernel to load the address of unsafe instructions in debug registers. With this monitoring system in place, any attempt to execute unsafe instructions will be vetted by the kernel, and the page is marked executable. If the debug registers were previously watching another page, that page is returned to pending executable status. This mechanism prevents unsafe PKRU-writing instructions without the need for binary rewriting.

At startup, the trusted loader registers the virtual address space of each library in T at runtime, and subsequent calls to `mmap`, `mprotect`, or `pkey_mprotect` are checked against the current PKRU value by the modified kernel. This interception guarantees that component memory accesses are consistent with their assigned trust levels.

Chapter 3

Attacks on PKU-based Isolation

After examining the kernel security model for PKU and the available attack surface in the sandboxes, we identified several potential vectors for circumventing the protections intended by intra-process isolation. We then developed ten concrete proofs-of-concept, listed in Table 3.1. The following sections develop a rough taxonomy of the attacks and present their technical details.

3.1 Methodology

3.1.1 Threat Model

We use the same threat model described in current research on secure isolation with PKU. We assume the attacker can execute arbitrary machine code in the untrusted domain, with the exception that executed code cannot initially contain unsafe `wrpkru` and `xrstor` instructions. We assume trusted component's interface is free of exploitable vulnerabilities. This is consistent with the threat model shared by both ERIM and Hodor.

We assume the initial state of the application is not compromised. The kernel, linker, or application is trusted to correctly initialize the PKU sandbox. Trusted components loaded from disk are assumed to be trustworthy (e.g., protected by file permissions).

[0]Portions of this chapter appeared in Usenix Security 2020.

Table 3.1: Summary of developed proof-of-concepts.

Attack Name	Key Syscalls
VM Read	`process_vm_readv`
Procfs Write	`open`, `seek`, `write`
File Mapping	`open`, `mmap`, `write`
Shared Memory	`shm_open`, `mmap`
Remap	`mmap`, `mprotect`, `mremap`
Sigreturn	`rt_sigreturn`
Map Race	`clone`, `mmap`
Scan Race	`clone`, `mmap`
Pkey	`pkey_mprotect`
Seccomp	`prctl`

However, after the sandbox initialization, we make no further assumptions about the code running in the untrusted component. In particular, the untrusted component may contain memory corruption vulnerabilities that allow an attacker to carry out a control flow hijacking attack and cause arbitrary behavior. While other mitigations aim to prevent control flow hijacking [75, 70, 61, 80], they also carry a significant performance penalty and may not be completely effective [22, 78, 76, 33]. Both ERIM and Hodor are designed to provide secure isolation without additional protection from control flow hijacking.

We assume no vulnerabilities in a trusted library, the kernel, or hardware. The kernel is assumed to be trusted and free of vulnerabilities. Similarly, we do not consider attacks that exploit flaws in hardware such as transient execution attacks [45, 19, 51].

3.1.2 Approach to Sandbox and Kernel Analysis

Because our threat model intends to protect against an attacker running arbitrary code, the attack surface consists of all system calls that are both unprivileged and that are not already restricted by the existing PKU-based sandboxes. We exclude privileged system calls because current intra-process isolation systems do not address the question of running with elevated system-wide privileges (i.e., as the root user) and do not appear to be designed for this scenario.

We examined kernel documentation, code, and communications on developer mailing lists. We manually reviewed each system call available on the x86-64 architecture in Linux 4.9 for any system calls that could affect a process's own virtual address space, memory contents, or other intra-process resources. After identifying these system calls, we consulted code and documentation to determine if they were able to undermine the security of the PKU-based sandbox. Publicly-available archived kernel developer mailing lists also offered insight into the *intents* of kernel maintainers, which allowed us to identify the difference between sandbox and kernel developers' views of PKU.

3.1.3 Attack Evaluation and Proofs-of-Concept

Based on the designs of the proposed PKU-based sandbox, we develop several distinct software attacks that allow an untrusted component to access protected memory. We evaluate our proposed attacks against the publicly-released source code of the ERIM project [71]. We tested our exploits against ERIM's `ptrace`-based sandbox, which runs in userspace and does not require kernel modifications. We wrote a library that allocates protected memory and stores a secret using ERIM's API. *In all cases, the attacker's goal is to disclose protected memory.* We consider an exploit successful when code in the untrusted component is able to access memory that is isolated to the trusted component without entering the trusted context through a legitimate call gate.

We made small changes to ERIM's code to get a more complete working prototype. Specifically, we implemented the on-demand executable page inspection described in Section 3.4 of ERIM's design and extended the `ptrace`-based sandbox to provide minimal support for multi-threaded processes as described in Section 3.7 of ERIM [72]. For each change, we attempted to keep the implementation as closely aligned with the design as possible. We did not introduce additional weaknesses, but we also did not add new protections against attacks not considered in the design.

We tested ERIM on an AWS EC2 `c5.large` virtual machine instance provisioned for this research, which provides two cores of an Intel Xeon Platinum 8124M processor. The virtual machine ran the Debian 8 operating system with Linux kernel version 4.9.168. At the time of testing, this is the latest build of the same major/minor kernel version used in ERIM [72].

3.2 Attack Details

3.2.1 Subverting Memory Permissions

Both ERIM and Hodor rely on basic assumptions about the enforcement of memory page permissions. ERIM assumes that pages cannot be mapped with both executable and writable permissions at the same time, an abstraction sometimes referred to as "W^X" (Write exclusive-or eXecute). Since Linux by default *allows* pages to be simultaneously writable

and executable, ERIM also introduces a "trusted-only execute memory" (TEM) module to intercept `mmap` and `mprotect` calls and enforce W^X. When the application requests a new executable page, the TEM module takes one of two possible actions. If the call originates from the trusted domain, then the request is allowed unchanged. Otherwise, the TEM module removes the executable permission before forwarding the request to the kernel, but the TEM module internally records that page as pending executable. When the process attempts to execute that page, the kernel delivers a segfault signal to the process. The TEM module handles the signal, checking if it originates from an attempt to execute a page that was requested with execute permissions earlier. If so, the TEM module scans the page for `wrpkru` or `xrstor` instructions. Upon determination that the page is safe, then the TEM module instructs the kernel to mark the page as executable *but not writable*. The TEM module is intended to preserve two critical properties: 1) The untrusted domain cannot mark unsafe pages executable, and 2) The untrusted domain cannot give a page writable and executable permissions at the same time.

Hodor takes a similar approach, introducing kernel patches that add new checks to some memory-related syscalls and inspect executable code for `wrpkru` and `xrstor` instructions. Hodor also currently prevents executable code from being mapped writable, although the authors describe a possible extension that allows code pages to be safely modified and inspected using a mechanism analogous to ERIM's segfault handler.

It is *critical* that the untrusted domain does not have access to a page that is both writable and executable. If the untrusted domain were able to write directly to executable memory, then it could simply write an unsafe `wrpkru` gadget and execute it. While in theory it would be possible to intercept and check every memory write using dynamic instrumentation, this approach would have an unacceptable performance impact. Instead, ERIM and Hodor use page table permissions as the hardware-supported mechanisms to prevent a process from writing and executing memory.

Unfortunately, both systems incorrectly assume that marking a memory mapping as non-writable makes the memory actually immutable. Surprisingly, in modern Linux kernels, the fact that a memory page is mapped without writable permissions does not guarantee that the memory is immutable. We developed several proof-of-concept attacks that exploit this

faulty assumption to execute arbitrary unsafe code and gain control over the PKRU register from an untrusted domain. Similarly, we found multiple interfaces that Linux, by design, provides for accessing process memory that ignore PKU domains on a page.

Linux provides several interfaces that allow processes to access their own memory indirectly, through the kernel. In many cases these interfaces bypass checks for page read/write/execute (`rwx`) permissions, PKU permissions, or both. Any interface that bypasses page write permissions can modify the code of the process at run time to add an unsafe `wrpkru` instruction that unlocks all PKU domains.

Inconsistent Enforcement of PKU Permissions

The `process_vm_readv` and `process_vm_writev` syscalls both provide a kernel interface through which a process can read and write the memory of a target thread. These calls require no privileges (and in fact bypass LSM checks) when the target thread is in the same thread group (process) as the calling thread. Additionally, neither proposed PKU-based sandbox traces or restricts these calls. Therefore, a process is always allowed to access its own memory via these syscalls. Documentation for `process_vm_readv` and `process_vm_writev` states that they will fail if they attempt to access memory "inaccessible to the local process," [3] but this documentation is ambiguous in the context of PKU permissions—is memory blocked by the current state of the PKRU register considered "inaccessible?" In testing, we found that these calls do respect traditional page permissions, but ignore PKU domains. An untrusted application can therefore use these syscalls to access memory that would otherwise be protected by the PKU system.

While this issue is an oversight in existing implementation, it is not difficult to fix. The sandbox can inspect calls to these syscalls and deny access to PKU-protected pages from untrusted application components. Since these calls seem to be never or rarely used in common applications this would have negligible performance impact. No references to these calls appear at all in the source code of the applications benchmarked in ERIM.

Inconsistent Enforcement of Page Table Permissions

In addition to kernel interfaces that merely ignore PKU protections, there are also interfaces that deliberately allow processes to read and write memory regardless of page table permissions *or* PKU tags. The `ptrace` syscall allows reading and modifying memory without being subject to page permissions *or* PKU permissions, and a thread is always allowed to attach to another thread in the same thread group. In this way, an untrusted application can modify executable code even in a non-writable page to add unsafe `wrpkru` instructions, or simply read the PKU-protected memory directly, regardless of the current state of the PKRU register. This attack may not be possible against ERIM's `ptrace`-based sandbox because an application cannot be traced twice, but it would be possible against kernel-based sandboxes such as Hodor or ERIM's kernel TEM module. This attack could also be prevented just by limiting calls to `ptrace`, again with negligible performance impact for applications that do not frequently call `ptrace`.

The most problematic alternative interface is in the `mem` pseudo-file provided by `procfs`. This file supports standard IO operations via the usual syscalls (open, seek, read, write), but treats the file stream position as an *address* in the process's virtual address space. A process can open its `mem` file at the path "`/proc/self/mem`", seek to an arbitrary offset, and perform reads or writes at that address. Reads and writes made through this interface, by design, ignore permissions on page mappings. An untrusted application can either read protected memory directly from this interface, or modify unwritable code in order to control the PKRU register.

This interface is more difficult to restrict without either making changes to the kernel or significantly impacting performance, since using the ptrace-based sandbox would, at a minumum, require tracing every `open`-like syscall. Unlike the `mmap`-like calls that are currently tracked by the ptrace-based sandbox, `open`-like calls are very common in typical applications, as supported by our performance analysis in Section 3.4. Removing the "`/proc/self/mem`" file would require kernel changes and might break compatibility with programs that use this file for legitimate purposes.

Mappings with Mutable Backings. Another problem arises when processes can map memory into their virtual address space that is *backed* by something mutable even though the mapping may be marked non-writable. Recall that page permissions (and PKU tags) are associated with the virtual memory mappings, not with the object that the mapping refers to. In this case, it is possible for an attacker to create an executable, non-writable mapping to memory that contains no unsafe `wrpkru` instructions initially, but is backed by a mutable object. The non-writable permission prevents modifications made to the memory through that mapping, but it does not prevent the underlying object from being changed. This allows the attacker to modify the underlying object to add an unsafe `wrpkru` (and execute it) without detection by the sandbox. Figure 3.1 illustrates two examples of this class of attack.

The simplest example is a memory-mapped file. The `mmap` syscall allows the caller to specify a file descriptor, which will then expose a given portion of a file as memory in the caller's virtual address space. Even if the mapping is made without write permissions, the file system permissions of the backing file may be writable. Any changes made to that file are then reflected in the process's view of that memory as well. So, an attacker running code in the untrusted domain can create a file with `rwx` file system permissions in any writable location (e.g. `/tmp`) and write some innocuous code to the file. The attacker will then map the file in virtual memory with `r-x` permissions using `mmap`, but write an unsafe `wrpkru` gadget to the file using the `write` syscall. Finally, the attacker can execute the `wrpkru` gadget to unlock all PKU domains.

A similar attack is possible without touching the file system by using a shared memory mapping. In Linux, processes can create or obtain a reference to a shared memory object with the `shm_open` syscall. The shared memory can then be mapped into the process virtual address space via the standard `mmap`. There is no requirement that page permissions be consistent across multiple mappings of the same shared memory, either across or within processes. For example, it is possible for a process to map the same shared memory page into its virtual address space twice: once with `r-x` permissions and once with `rw-` permissions. Any changes made by writing `rw-` page are reflected in the `r-x`, since both mappings refer to the same physical memory. Even if the sandbox were able to prevent a process from mapping

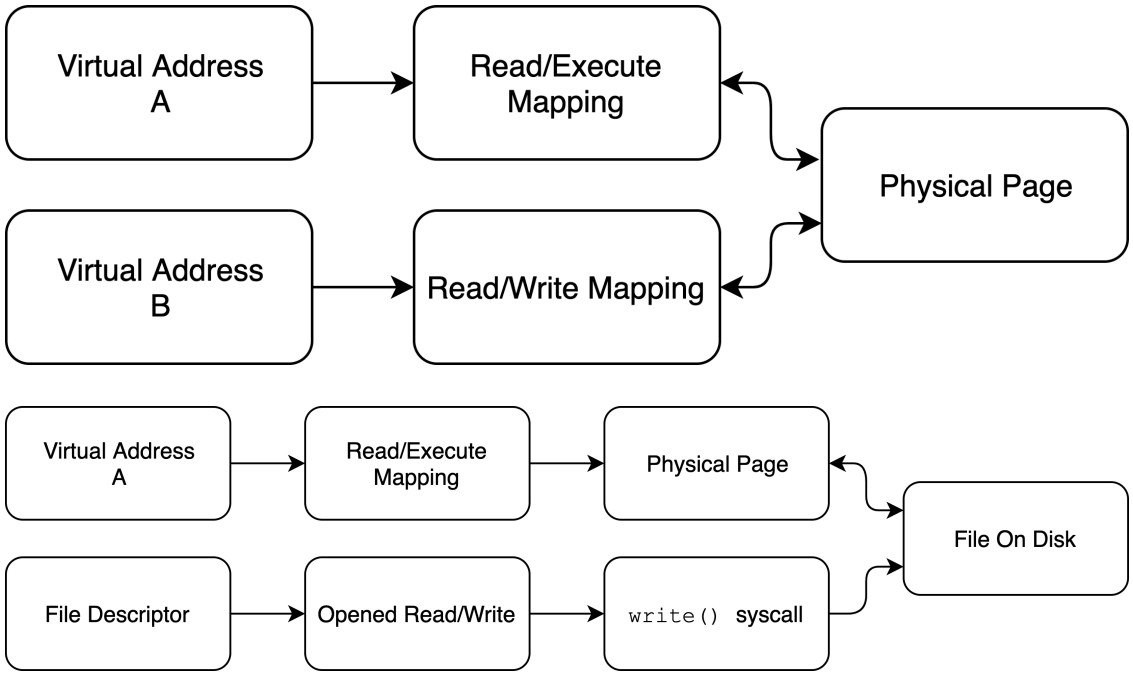

Figure 3.1: Two examples where W^X does not guarantee code immutability in Linux. Permissions applied to virtual memory mappings do not necessarily apply to the underlying physical memory or file that backs the mapping.

the same memory twice with different permissions, the same attack is possible as long as the attacker can fork a separate process and map the shared memory once into each process. To fully prevent this attack, significant restrictions would have to be placed on shared memory in general (such as disallowing executable mapping of all shared memory). Alternatively, a kernel-wide state could be kept in order to prevent the same memory from being mapped twice in any process with incompatible write and execute permissions.

3.2.2 Changing Code by Relocation

The previous attacks all read or write memory that was assumed inaccessible due to page-level permissions. When memory that is not expected to be writable can be modified, an attacker can introduce dangerous `wrpkru` gadgets to executable memory. However, it's also possible for an attacker to introduce `wrpkru` instructions just by changing the *locations* of memory mappings, without changing their content. The `mremap` system call allows the attacker to move pages to different locations in the address space, but current PKU-based sandboxes do not intercept this syscall. Because x86-64 instructions are not aligned, just rearranging pages can create instructions that did not exist before, including `wrpkru` instructions.

Concretely, an attacker can exploit this by creating two memory mappings at distant addresses that each contain *part* of a `wrpkru` instruction at the page boundary: the first half of the instruction bytes at the end of one page, and the ending half of the instruction bytes at the beginning of the other page. At the time the pages are mapped in, the sandbox vets each mapping for unsafe `wrpkru` instructions. Because neither page contains a complete instruction, both mappings are allowed. The attacker then calls `mremap` to move the pages into an adjacent position in the virtual address space. Since the sandbox does not monitor or restrict `mremap` calls, the attacker successfully creates a new `wrpkru` gadget without interception by the sandbox. Figure 3.2 visualizes this attack.

This attack shows that it is not sufficient for a sandbox to inspect calls that create new memory mappings or modify mapping permissions; the sandbox must inspect any call that might modify the arrangement of mapped pages as well. Whenever the virtual address space

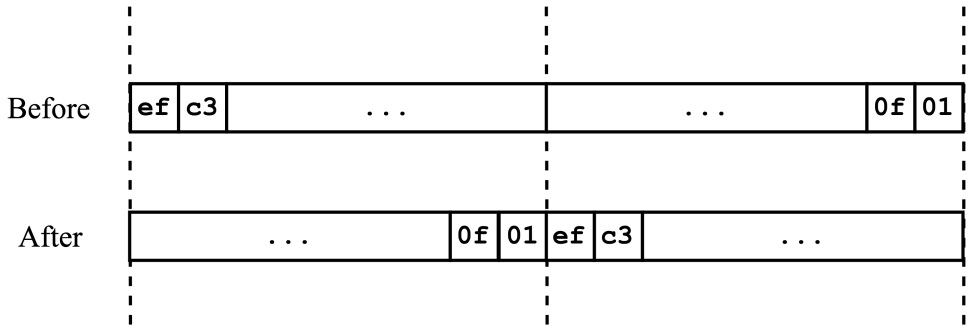

Figure 3.2: The `mremap` attack. Two pages of code are initially mapped into the process. At this point, the bytes for the `wrpkru` instruction out of order, so the mappings are allowed by the sandbox. A call to `mremap` modifies the page layout to introduce a new `wrpkru` gadget, but does not trigger a new scan.

of the process changes, the sandbox must re-scan the boundaries of any affected executable pages to maintain its security invariants.

3.2.3 Controlling PKRU from the Kernel

Both ERIM and Hodor focus on ensuring that there are no useful `wrpkru` or `xrstor` gadgets available to an attacker in the untrusted application, but the kernel can also modify the state of the PKRU register. Therefore, PKU-based sandboxes must also consider kernel interfaces that may allow a process to control the PKRU value indirectly.

The `sigreturn` syscall provides a concrete example of a kernel interface through which an attacker can modify the PKRU register. Previous work by Bosman and Bos [18] showed that a single `sigreturn` gadget is enough for an attacker to execute Turing-complete code or to make arbitrary syscalls without introducing new machine code, and that such gadgets are widespread in real-world systems. We find that a `sigreturn` gadget also allows the attacker to control the PKRU register without needing a `wrpkru` or `xrstor` instruction.

A process ordinarily uses `sigreturn` to restore the program's execution state after handling a signal. When the kernel delivers a signal to the process, it first stores the process's execution state on the stack of the signal handler. It then pushes a return pointer to a `sigreturn` trampoline and starts execution at the signal handler. When the handler returns, it pops the return pointer to the `sigreturn` trampoline. The trampoline then makes the `sigreturn` syscall, with the previously-stored state still on the stack. Figure 3.3 illustrates the state of the userspace stack upon signal delivery.

Inside the kernel, `sigreturn` restores the process CPU state from the stored values on the stack before returning to userspace. That state includes the contents of registers such as the instruction pointer, stack pointer, and general-purpose registers. It can also contain an extended set of registers including floating-point registers and the PKRU register.

Since existing PKU-based sandboxes only consider `wrpkru` and `xrstor` instructions, they do not prevent the untrusted application from modifying its own PKRU register via `sigreturn`. An attacker can set up a crafted state on the stack and make the `sigreturn` syscall to convince the kernel to "restore" an arbitrary value to the PKRU register without needing a `wrpkru` or `xrstor` gadget in userspace.

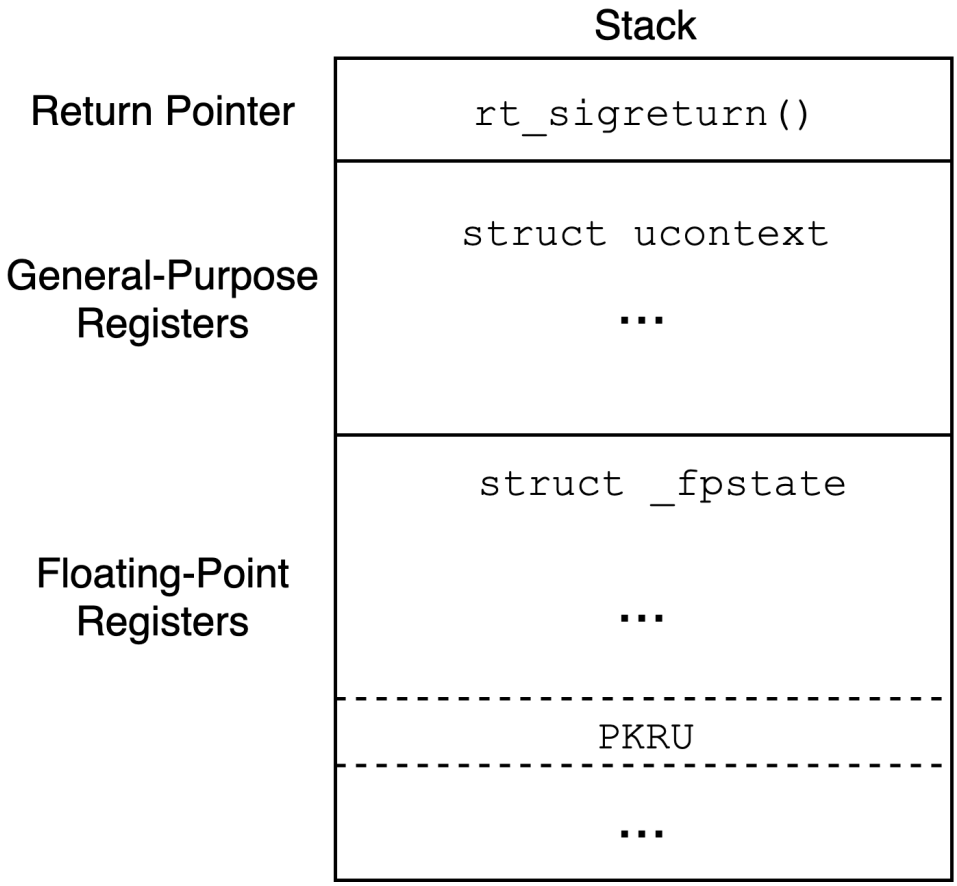

Figure 3.3: State of userspace stack during signal delivery. The saved PKRU state is stored among the CPU extended state in unprotected memory, and can be modified by the handler or another thread before it is restored by the kernel.

Note also that proposed patches [16] to the Linux kernel that mitigate the `sigreturn`-oriented programming attacks described by Bosman and Bos [18] do not appear to prevent this attack from working against PKU-based sandboxes because they are aimed at preventing initial exploitation of `sigreturn` calls by an attacker to bootstrap a control-flow hijacking attack. In contrast, the threat model for intra-process memory isolation assumes that an attacker already controls execution in the untrusted component. The proposed patches make blind exploitation of `sigreturn` gadgets more difficult by requiring a secret "signal cookie" placed by the kernel at signal delivery to remain intact upon signal return. This mitigation does not stop an attacker who already has the ability to register signal handlers or arrange for the delivery of real signals.

3.2.4 Race Conditions

The architecture of PKU-based memory isolation sandboxes must also consider an attacker who can attempt to exploit race conditions by controlling more than one thread. An attacker who compromises the control flow of one thread can generally hijack other threads by tampering with their stacks. Even in an application that is ordinarily single-threaded, an attacker can call the `clone` syscall to create a new thread. Consequently, sandboxes must either handle race conditions or explicitly forbid new threads by blocking calls to `clone`.

Existing designs do consider some potential race condition attacks. ERIM specifies that the trusted library T should allocate a PKU-protected stack to prevent other threads from accessing intermediate data or hijacking control flow while T is executing. Hodor also requires that each trusted library has its own set of stacks that are accessible only from that library. However, there are other attack vectors for race conditions that must also be carefully considered.

Signal Delivery

Hodor additionally blocks delivery of signals while the trusted library is executing, in order to prevent an attacker from interrupting the trusted library. Recall from Section 3.2.3 that

signal delivery stores CPU state including general-purpose registers on the stack. This means that if a signal is delivered while execution is in the trusted component T, the kernel may leak the contents of T's registers to the untrusted application by placing them on an unprotected stack. Note that this issue is distinct from the ability to control PKRU via `sigreturn`; this information leakage would occur at the time of signal *delivery*, not the return from the handler.

At first glance it may appear that this issue could also be easily fixed by using a PKU-protected stack for signal handling. In Linux, a process can use the syscall `sigaltstack` to specify a memory region to be used as the stack for signal handling. However, when the kernel delivers a signal it first writes the context data to the handler stack before transferring control to the handler. The kernel checks the value of the PKRU register at the time of signal delivery. If the write would not be allowed under that PKRU value, then the kernel refuses to write the context data and instead delivers a segfault to the process. This design choice by the Linux kernel developers makes it difficult to set up a protected stack for signal handling that functions correctly regardless of when a signal is delivered.

Memory Scanning

Recall that both ERIM and Hodor scan new executable memory that is loaded by the untrusted application to vet it for unsafe `wrpkru` gadgets. To do this securely, it is critical that the order of operations is considered. For example, consider an application that makes an `mprotect` call to change the permissions on one page from `rw-` to `r-x`. If the order of these operations is not handled carefully, then it may leave the implementation vulnerable to one of two race conditions. First, if the sandbox performs the scan making the permissions change, then a second thread in the untrusted application can modify memory during the scan but before the permissions change. This may result in code that was safe at the time it was scanned, but is not safe by the time it is marked executable. If instead the sandbox makes the permission change first and then does the scan, then another thread can attempt execute the unvetted page before the scan completes.

Several factors make this race condition practical to exploit. First, the attacker may fork child processes to repeatedly attempt the race condition. Secondly, the attacker can get feedback (via the output of the child process) on whether an attempt failed because the change was made *too early* (before the bytes were scanned) or *too late* (after the page was no longer writable). Some amount of CPU scheduling is also under the attackers control; in a multi-core system, the attacker may bind each thread to separate cores to increase the odds that they run concurrently. The combination of these factors allows an attacker to repeatedly attempt the exploit, while dynamically adjusting the timing based on feedback from each attempt.

In order to close off this avenue of attack, the sandbox must temporarily render the page both non-writable and non-executable, perform the scan, and then mark the page readable and executable. This ordering prevents both execution of the memory before the scan and modification of the memory during the scan, assuming that attacks subverting page permissions (detailed in Section 3.2.1) are also mitigated.

Determination of Trusted Mappings - ERIM

The paper describing ERIM does not detail exactly how the ptrace-based sandbox determines whether a mapping is requested by the trusted library T or not. However, the published implementation uses a bit stored in global (PKU-protected) memory to enable trusted mappings. The bit is set just before performing a trusted mapping and cleared immediately afterwards. When a memory-related call is made, the tracer reads this bit to determine if the process is currently in a trusted context. Note that since the bit is stored in a PKU-protected page, the untrusted component cannot simply toggle this bit itself to perform a trusted mapping; it must go through the trusted component.

However, since this flag is shared for the whole process, a race conditions results when there are multiple threads mapping memory. If an untrusted context makes a mapping at the same time that a trusted mapping is being made, then the both mappings will be accepted as trusted.

```c
// Trusted Component

char *secret;
void allocate_secret() {
  // map a new secret page
  secret = mmap(NULL, PAGE_SIZE, read_write, ...);

  // isolate the page to the component
  int r = pkey_mprotect(secret_page,
    PAGE_SIZE,
    read_write,
    TRUSTED_PKEY);

  // If pkey_mprotect fails, kill process
  if (r != 0) exit(1);
}


// Untrusted Component

void exploit() {
  // override the pkey_mprotect syscall to skip but return 0
  add_seccomp_override(SYS_mprotect_key, 0);

  // call trusted library - pkey_mprotect() silently fails
  allocate_secret();

  // secret is accessible to untrusted component
  printf("secret: %s\n", secret);
}
```

Listing 3.1: C-like pseudocode illustrating how seccomp filters can be used to manipulate intra-process behavior.

The attacker can create a second thread that attempts to create a mapping with `rwx` permissions in a tight loop. On the main thread, the attacker induces a legitimate trusted mapping call by mapping in new, *safe* executable code. After the subsequent scan completes, ERIM's trusted library makes a trusted `mprotect` call to mark the code as `r-x`. Meanwhile, the second thread is repeatedly attempting to make `rwx` mappings. In an untrusted context, these mappings are ordinarily downgraded to `rw-` by the tracer. However, when the call is made simultaneously with the legitimate trusted mapping in the main thread, the tracer checks the global variable and mistakenly identifies both as trusted. This allows the `rwx` mapping to succeed. Once the mapping has been created, the attacker can write and execute unsafe code on this page without detection by the sandbox. Figure 3.4 shows the execution of both threads on the same timeline. The attacker repeatedly attempts to create `rwx` in a very small loop, while another thread creates legitimate trusted `r-x` mappings. Afterwards the attacker checks the permissions of the mapped page and uses any that are `rwx`. Because the attacker can quickly and repeatedly attempt the race condition without any adverse effects, exploiting this race condition is very practical.

3.2.5 Interfering with Non-memory Shared Resources

Besides attacking memory directly, an attacker may also target other process-wide shared resources that may affect the behavior of the trusted library. Consequently, even trusted libraries with apparently bug-free code may have vulnerabilities when they rely on assumptions about resources that may be open to tampering from the untrusted components.

Influencing Intra-process Behavior with `seccomp`

One example of a potentially exploitable shared resource is the seccomp filter associated with the process. Processes can specify a seccomp filter via the `prctl` syscall. The filter runs each time the process attempts to make a syscall, and may either allow the syscall to execute or cause it to return immediately with a specified value. ERIM's ptrace-based sandbox uses a seccomp filter to intercept memory-related syscalls, but this does not stop the untrusted component from further installing new filters. When multiple filters are installed, all are run

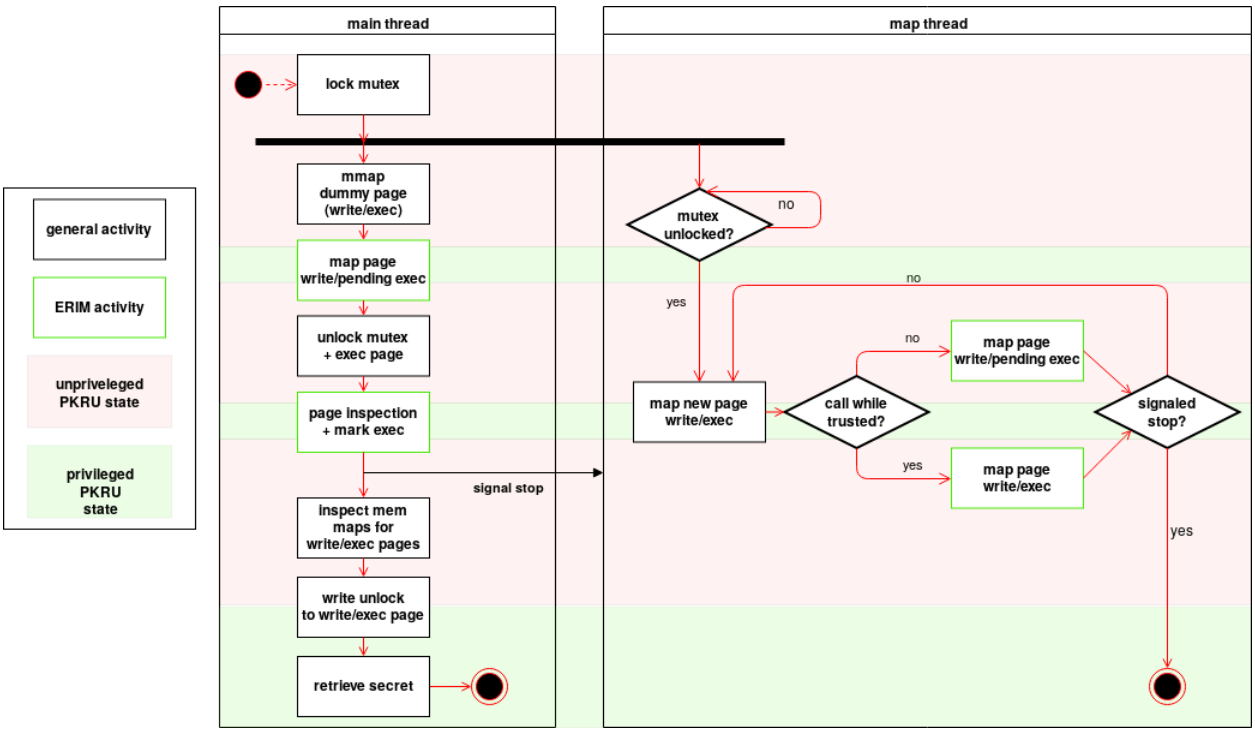

Figure 3.4: In ERIM, a race condition allows an untrusted thread to make writable and executable mappings, as long as they are made concurrently to a trusted component mapping in another thread.

but certain return results take precedent over others (in general, the more restrictive result takes precedence) [4].

A malicious seccomp filter can alter the behavior or return values of a syscall in a way that violates the ordinary behavior of the syscall, creating an exploitable condition in otherwise correct code. Linux does not allow a process with a seccomp filter to execute an SUID application in order to prevent it from undermining the behavior of the application run with elevated privileges [4]. The same risk applies when switching to a trusted component in a PKU-based secure memory isolation system.

More concretely, imagine a trusted library that allocates new memory and then tries to protect the memory with its PKU domain by calling `pkey_mprotect` on it. The library trusts the kernel to execute the call and update the PKU domain on the mapped memory, or return an error value. However, a malicious seccomp filter could deny `pkey_mprotect` calls and force them to return a value indicating success. This attack would allow an attacker to trick the trusted library into using unprotected memory that it believes to be isolated. Listing 3.2.4 demonstrates this attack in C-like pseudocode.

Modifying Trusted Mappings

An attacker may also try to access isolated memory or modify the trusted library code by changing the virtual address space of the trusted library. Hodor discusses such attacks and prevents them by informing the kernel of the trusted libraries code and data addresses, then preventing further attempts to change those mappings from an untrusted context. However, ERIM does not consider such attacks.

For example, instead of trying to change the PKRU register to grant access to a particular PKU domain, the attacker may simply change the PKU domain associated with the mapping to make it accessible. The attacker can make a `pkey_mprotect` syscall, changing the protection key on any page to the untrusted domain. The kernel allows this call regardless of the PKRU register state of the caller or the PKU domain of the targeted memory; there is no requirement that the caller is able to actually read or write the targeted memory at the time the call is made. Because trusted component has permission to access *both* the trusted

and untrusted domains, subsequent accesses from the trusted component succeed as usual, and the trusted component is unaware of the change in the page's protection key.

Similar attacks are possible by targeting the code mappings of a trusted library. If the code that immediately follows a trusted context switch can be swapped out using syscalls like `munmap`, `mmap`, or `mremap` then the integrity of the trusted library code may be compromised.

3.3 Discussion

The diverse set of vulnerabilities in existing PKU-based sandboxes require diverse mitigations. Attacks that take advantage of alternate memory access paths (detailed in Section 3.2.1) require a comprehensive solution to guarantee the integrity of executable code in a process's virtual address space, which we discuss later in Section **??**. Race conditions can be mitigated by a design that incorporates a multi-threaded attacker into the threat model and orders operations carefully, preventing intervening changes, which we also discuss later in Section 5. Other attacks, such as the `mremap` and `seccomp` exploits described in Sections 3.2.2 and 3.2.5 can be mitigated by more carefully restricting certain syscalls. However, tracing additional syscalls in the `ptrace`-based sandbox has a high overhead for syscalls that are called often, which we measure in this section. Still others attacks, like the `sigreturn` attack in Section 3.2.3, could be easily mitigated by kernel changes but do not appear to be completely fixable using only the `ptrace`-based sandbox architecture.

Intra-process isolation fundamentally changes the threat model of an operating system that otherwise gives processes a high degree of control over their own code and environment. As a result, intra-process isolation systems risk turning otherwise innocuous kernel interfaces into vulnerabilities. PKU-based memory isolation systems are especially fragile because the Linux kernel does not treat PKU as a security feature (in fact, a recent patch to the Linux kernel introduced an internal kernel helper function for bypassing PKU-related checks on userspace memory accesses [37], which is used to service some syscalls such as `process_vm_read`). In this situation, it is difficult for system designers to conclusively identify every kernel interface that may violate the new security assumptions imposed on it.

3.3.1 PKU: Reliability or Security?

PKU is not designed as a security feature, since an unprivileged instruction can assign arbitrary rights to the PKRU register. Of course, this design does not mean that PKU cannot be repurposed for security. But it does have important implications for the way kernel developers perceive and treat the feature. For example, an early discussion on a Linux developer mailing list (which decided how `sigreturn` should treat the PKRU register) envisioned PKU being used to provide *reliability* against accidental out-of-bounds memory accesses, rather than *security* against an intentional attacker [55]. Developers likely used similar reasoning when deciding to allow processes from indirectly accessing their own PKU-protected memory through interfaces like `process_vm_readv`, `ptrace`, and `/proc/self/mem`.

ERIM and Hodor must assume a trusted, secure kernel. Otherwise, whatever security guarantees they provide in userspace are moot. However, the above issues show that it is not enough for the kernel to be trustworthy in its own threat model, but that it must also enforce the new trust boundaries required by the PKU sandbox. The PKU-based sandboxes try to augment the kernel to this end, but it is very difficult for the sandbox designers to retroactively find and undo the large number of security-relevant decisions that kernel developers made when supporting the PKU feature without the expectation that it would be used for security.

3.3.2 Assumptions in Secure System Design

The PKU-based sandboxes that we examine in this paper both suffer from similar vulnerabilities because they make some of the same incorrect assumptions, particularly assumptions around the kernel's management of the PKU feature and virtual memory. They assume that, by preventing writable executable memory mappings, W^X fully protects a process's code. This abstraction of W^X in Linux is also used in other systems security papers that rely on code integrity [32]. However, as our work demonstrates, the Linux virtual memory system requires non-trivial changes to achieve robust code integrity.

3.3.3 Towards Mitigation

Virtual Address Space Integrity. The first challenge to completely addressing the vulnerabilities presented earlier is to mitigate attacks that undermine the virtual address space integrity of the process. Although Linux supports basic page permissions, it is not designed to support strong guarantees about the integrity of code (even in non-writable mappings), as evidenced by the several interfaces that intentionally circumvent page mapping permissions (detailed in Section 3.2.1). To close these loopholes, userspace applications need a way both to disallow further changes to non-writable pages from interfaces like `ptrace` and `procfs` and to prevent mappings that cannot guarantee integrity (i.e., shared memory or file-backed mappings where changes to the underlying resource may be seen in the mapped memory). In Chapter 5 we investigate wider implications of these findings and propose systemic solutions. In Section 3.4 we rule out the obvious solution of simply extending the `ptrace`-based sandbox by showing that it leads to unacceptable performance degradation.

The interfaces that ignore page permissions like `ptrace` and `procfs` are relatively straightforward to mitigate. Since the kernel currently intentionally allows access for these interfaces, patches could be introduced to deny access instead. Both of these interfaces are mediated by "`ptrace` access checks," which presently are universally allowed for same-PID accesses [6]. The kernel could simply add a new interface for userspace processes to request that further `ptrace` access checks are denied even for the same PID.

Race Conditions Associated with Seccomp-based Filtering. In addition to challenges securing the virtual address space, another serious problem for the `ptrace`-based sandbox architecture presented earlier in Section 3.4 is a race condition inherent in certain types of filtering with `seccomp` and `ptrace`. The `seccomp` filter language provides support for *numeric* filtering of syscall arguments in-kernel, but any further inspection (e.g., dereferencing pointer arguments) is possible only from the tracer running in userspace via the `ptrace` interface [4]. However, if the tracer does dereference a pointer and allow the syscall to proceed, then the memory is accessed twice: once from the tracer, and once from the kernel when the syscall is actually executed. Therefore, the tracer has no guarantee that the value inspected is unmodified by the time it is used.

This potential race condition makes it difficult for a `ptrace`-based sandbox to safely inspect arguments to syscalls that require the kernel to read from userspace memory, such as `open` (which accepts a pointer to the path string) and `sigreturn` (which reads a saved context structure from the processes's stack pointer). In both cases, the tracer may inspect the memory of the process, but if it finds safe values and allows the syscall to proceed then it has no guarantee that the memory is unchanged when the kernel accesses it.

3.4 Performance Impact of Extended Ptrace-based Sandboxing

We develop an extension to the `ptrace`-based sandbox that prevents a *subset* of the attacks developed earlier. Notably, we can partially mitigate the exploits from Section 3.2 by tracing added system calls in ERIM [72]. These system calls are shown in Table 3.2, along with what threat vectors they mitigate. We add additional seccomp BPF filters to the `ptrace`-based sandbox module of ERIM, which routes calls to ERIM that need to be checked for memory access permissions. ERIM's `ptrace`-based sandbox runs only in userspace and does not require kernel changes. The `ptrace`-based sandbox instruments programs by calling them with a binary provided in the ERIM software package. This sandbox model is the more likely target for practical deployment of ERIM to protect real users against software vulnerabilities.

We re-iterate that these additional traces *do not* constitute complete mitigations to the attacks described in Section 3.2 against PKU systems in general, but serve to demonstrate a lower bound on overhead to the proposed ERIM system when adding the necessary additional syscall traces to ERIM. We emphasize that these results apply only to the `ptrace`-based sandbox architecture, where performance is heavily dependent on the number of system calls requiring a context switch to the tracing process. Kernel-based solutions (for example, using Linux security module) avoid this performance problem but incur deployability and maintainability costs.

The authors of ERIM [72] measured the throughput of the popular NGINX webserver in requests per second using a server implementing OpenSSL with and without ERIM protecting

secure key access. This benchmark serves to illustrate the performance impact of a webserver protected by ERIM against software vulnerabilities versus a server that is not protected. The authors claim that ERIM achieves roughly 95% to 98% of the performance of the native, non-protected OpenSSL using the kernel module implementation of ERIM. We first replicate the 2% performance impact in requests per second shown by the ERIM Kernel bars in Figure 3.5.

We use the identical configuration to the published ERIM [1], on an PKU-enabled Amazon Web Services c5d.4xlarge EC2 instance, which has a 16-core Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz processor, 32 GB of RAM, and a 450 GB NVMe SSD. We run the benchmarks with a 1-worker NGINX, 5 iterations, and 120 seconds of measurement time per benchmark. We increased the iterations used to average the requests per second from 3 to 5 and increased the time from 65 to 120 seconds because these options from the published configuration yielded larger standard deviations. With these parameters, all following results had standard deviation percentages of less than 1.0.

To measure ERIM with the additional traced syscalls, we first need to examine the performance of ERIM in kernel mode versus using the `ptrace`-based sandbox module. The authors of ERIM claim that ERIM with ptrace has the same performance of the kernel mode ERIM with only 2% overhead. We measure the difference and show the comparison in Figure 3.5. Notably, we find the `ptrace`-based sandbox version **incurs a significant performance impact** at lower content sizes compared to ERIM running in the kernel. At 1kb of content fetched by the Apache Benchmark suite, ERIM in userspace suffers 20% worse performance than the published kernel mode ERIM benchmarks, and slowly approaches the native and kernel version as more content is fetched.

We then altered ERIM to filter the additional system calls shown in Table 3.2 and measured the performance. We find that modifying ERIM's `ptrace`-based sandbox to trace the syscalls responsible for the vulnerabilities in Section 3.2 results in a **40% greater loss in throughput on top of the published version from August 2019**. In raw performance numbers, this loss in throughput translates to NGINX operating at 76,545 requests per second for native performance at 1kb of content, 74,413 requests per second for the original ERIM performance at 1kb of content, and 29,728 requests per second when ERIM has the

[1] https://gitlab.mpi-sws.org/vahldiek/erim/tree/master/bench/webserver

Table 3.2: Additional syscalls traced by our modified ERIM in order to demonstrate the performance impact of only one portion of the patches needed to secure memory isolation with PKU instructions.

Additional Traced Syscall	Comment	Mitigates Attack
`open`	Opening files or file-like objects	Procfs Write/File Mapping/Shared Memory
`creat`	Functions like `open`	Procfs Write/File Mapping/Shared Memory
`openat`	Functions like open()	Procfs Write/File Mapping/Shared Memory
`munmap`	Additional mmap-like call	Modifying trusted mappings
`mremap`	Additional mmap-like call	Remap
`remap_file_pages`	Moves file-backed pages	Remap
`prctl`	Modifies process properties	Seccomp
`ptrace`	Traces processes	Indirect memory access
`process_vm_readv`	Reads memory from other processes	VM Read
`process_vm_writev`	Writes memory from other processes	VM Read
`sigaltstack`	Signal handling	Prevents changing signal handler
`rt_sigreturn`	Processes signals	Sigreturn

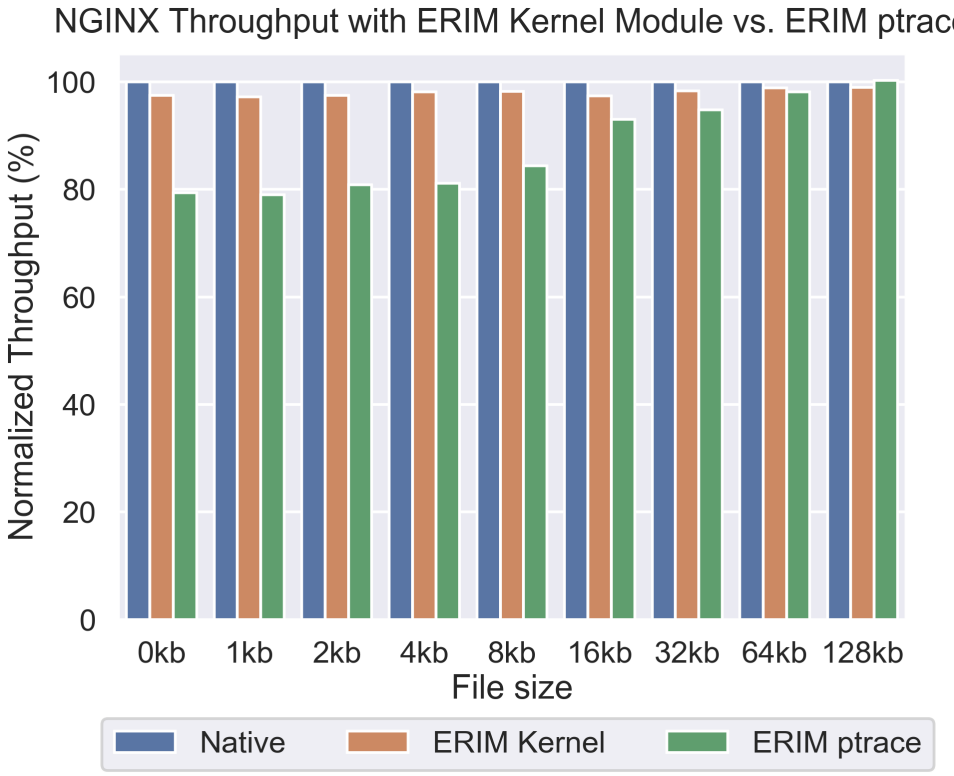

Figure 3.5: NGINX Throughput (requests/second) with one worker, normalized to native (no protection), ERIM kernel mode vs. ERIM with ptrace.

additional traces applied to mitigate the attacks in Section 3.2. ERIM-based web servers including the additional traces identified in Table 3.2 operate at only 40% of the throughput of the unsecured non-ERIM webserver performance, a stark difference from the 2% claimed by Vahldiek et al. [72].

Figure 3.6 also highlights a version of ERIM where the additional traces are restricted to only the `open` syscall. By examining the performance impact of only the open call, we reveal that ERIM's tracing of `open` alone leads to much of the loss in performance seen for all syscalls traced in Table 3.2. The performance overhead of tracing `open` serves as a lower bound to mitigate the Procfs Write, File Mapping, and Shared Memory vulnerabilities from Section 3.2 and shown in Table 3.2.

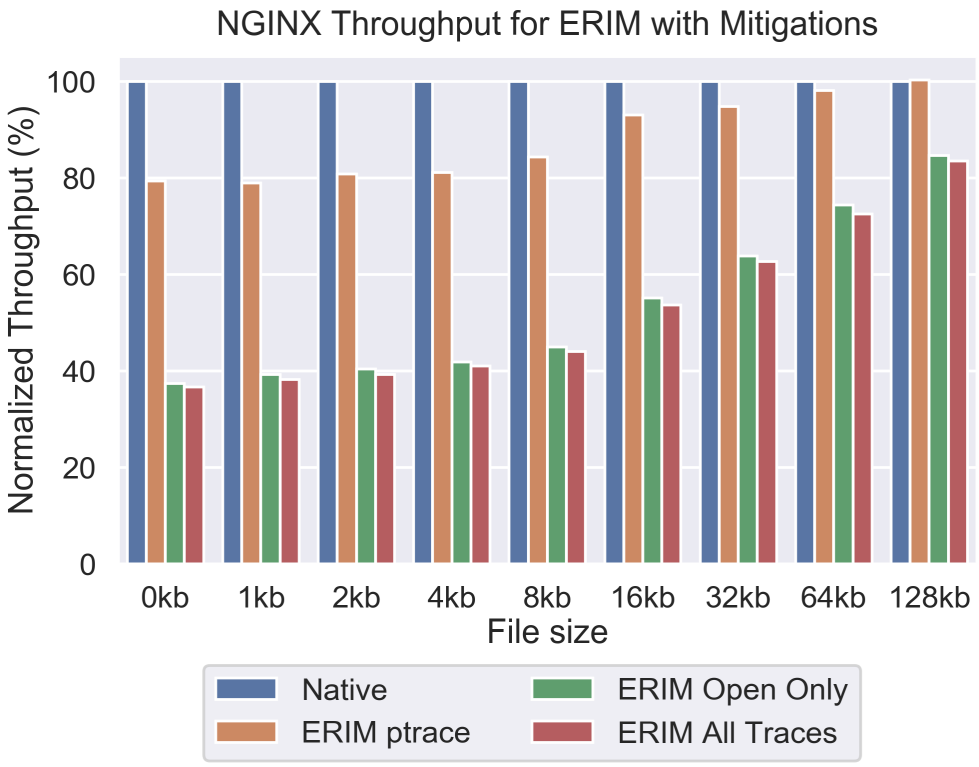

Figure 3.6: NGINX Throughput (requests/second) with one worker, normalized to native (no protection) of the original ERIM, ERIM with the open() call traced, and ERIM with all syscalls from Table 3.2 traced, with varying request sizes. Std. deviations all under 2.0%.

Chapter 4

Generalizing Attacks on Weak Memory Permission Model

In Section 3.2 we described several attacks against PKU-based intra-process isolation systems that subvert assumptions about W^X. Since this assumption is ubiquitous in systems security literature, a natural question is whether any of these attacks apply to other in-process protections like control-flow integrity (CFI). It is not immediately obvious that such a generalization is possible. PKU-based sandboxes are unique in that they allow the attacker nearly unrestricted access to execute machine code and system calls. In contrast, CFI intends to prevent control-flow hijacking in the first place. Still, we find that the `proc/mem` attack is highly general and can be exploited to break W^X assumptions in fully-precise control-flow integrity systems with shadow stacks under realistic assumptions.

We show that, because of gaps in memory permission enforcement in Linux, data-only attacks can sometimes be extended to overwrite application code in-memory. These attacks are feasible even in the presence of strong protections such as W^X, fine-grained static control-flow integrity (CFI) enforcement, and shadow stacks. We demonstrate the attack using a simulated memory corruption model applied to the widely-used nginx HTTP server [10]. This does *not* indicate the presence of an actual vulnerability in nginx; instead, we modified a local copy of the nginx code to intentionally introduce a vulnerability in order to simulate a memory corruption bug.

Data-only attacks are known to be a realistic threat with potentially serious consequences, but they are generally limited compared to control-flow hijacking attacks. By targeting specific data, a variety of outcomes can be accomplished:

- Overwriting security-sensitive values such as an is_admin flag can elevate an attacker's privileges within the scope of the application;
- Modifying file paths may allow the attacker to read or write local files;
- Controlling sensitive parameters to system calls like execve can allow the attacker to execute new programs.

An attacker may find that one of these outcomes is enough to achieve their ends. However, attackers may also be interested in objectives that are outside the immediate scope of the vulnerable application. They may want to gain persistent access to the exploited host, use the host to pivot into a private network, to gather data such as by sniffing network traffic on the host, to escalate privileges by exploiting additional vulnerabilities, and so on. Arbitrary machine code is the best possible outcome for an attacker because it grants them full access to perform any action with the application's privileges. It also serves as an ideal staging point to launch further attacks, such as exploiting a kernel or hardware bug for privilege escalation or pivoting through other hosts or networks.

Our contribution is to show how data-only attacks can combine with gaps in W^X protection in Linux to allow execution of arbitrary machine code. Previous techniques for executing code from data-only attacks required that the attacker control arguments to especially sensitive functions such as `system` or `execve`, or that the attacker could both write to the file system and force the loading of code from the same file. Unlike those techniques, ours requires only a file "offset-write" data-oriented gadget (a reachable code path that writes to an attacker controlled offset of an arbitrary file). Our method also does not rely on executing code through an interpreter (such as the shell), or writing or loading any code from disk. Instead it modifies machine instructions in-memory, making it more difficult to detect than attacks that modify write to long-term storage.

Crucially, we also demonstrate using data-only attacks to target and manipulate *high-level* application behavior rather than working at the individual function or instruction level.

By reasoning about the abstractions provided by the application and used in the code, we can reconfigure the application at runtime using data-only attacks. We use this technique to place the application in a state that is favorable to carrying out the `proc/mem` attack, regardless of its initial configuration.

The `proc/mem` attack allows arbitrary code execution in cases where it was not previously known to be possible. One would intuitively expect that controlled file write gadgets would be more common that e.g. `execve()` gadgets. Indeed, at least one previous case studies of vulnerable applications found that none of the 6 applications contained reachable `execve()` gadgets, while 3 allowed arbitrary file writes with CFI and shadow stacks. Of these 3, only 2 of these applications had functionality to load the attacker-written code from disk. The third, nginx, allowed arbitrary file writes but could not load the written code from disk. Morton et al [63] showed that data-oriented attacks against nginx are a serious realistic threat, demonstrating the ability to reconfigure the server to downgrade SSL versions, leak arbitrary files from the server, enable/disable access control, and more. Using a similar model, we additionally demonstrate that truly arbitrary code execution is also possible using the `proc/mem` attack.

4.1 Threat Model

Our threat model assumes that the attacker can repeatedly read and write application memory. This model requires more powerful vulnerabilities compared to the threat model used in related works like "Control-Flow Bending" [22] which permit the attacker only a single write of attacker-controlled data to an arbitrary location. However, it is not unrealistic. Use-after-free bugs frequently afford repeatable memory read and write exploit primitives from a single root cause, e.g. patched Google Chrome vulnerability CVE-2019-5786 [47]. Carlini et al. found that a single call to `printf` is enough for Turing-complete memory computation [22]. Hu et al. additionally showed that data-only attacks (on real-world vulnerabilities) can permit Turing-complete computation in process memory using a technique they termed data-oriented programming (DOP) [40]. Using DOP, an attacker can leverage a single bug to achieve Turing complete computation on application memory

(although DOP does not allow arbitrary interfacing *outside* the vulnerable application's address space as through system calls). The Turing-complete manipulation of memory in a DOP attack is equivalent to our model: the attacker can read memory, perform arbitrary computations on that data, and write data back to memory repeatedly.

We also assume that the attacker *cannot* cause the application to deviate from a feasible legitimate execution trace. In other words, the attacker is constrained to data-only attacks as defined by Carlini et al. This category of attack does not modify control data such as indirect branch targets or return addresses, with one exception: the attacker may write an indirect branch target as long as it is identical to a possible legitimate target of the call sites where it is used. For example, we allow the attacker to construct and write data structures that include function pointers for "handler" functions as long as one of the legitimate possible handler functions is used. In this specific sense, our model is a *more* restrictive type of attack compared to control-flow bending, which allows any control-flow transfer as long as the edge exists in the static control-flow graph. For example, control-flow bending allows a function to modify its own return address as long as the return target is another possible return site for that function. Control-flow bending thus allows *sequences* of control-flow transfer that are not possible in legitimate execution so long as each *individual* transfer corresponds to an edge in the legitimate control-flow graph. In contrast, data-only attacks only produce sequences of transfer that could appear in a legitimate execution. This property makes data-only attacks particularly hard to defend against, because they not only conform to any static CFI technology, but also to *dynamic* control-flow enforcement systems like shadow stacks.

4.1.1 Experimental Setup

We simulated this condition by writing a static nginx module that allows a remote attacker to read or write arbitrary memory addresses. Static nginx modules are added into the nginx binary at compile time. They therefore execute as part of the nginx binary with access to the same address space. Our module defines a handler for a specific URL, which allows the client to perform one of three actions:

1. `read`: read an arbitrary number of bytes from an arbitrary address.

2. `write`: write some bytes to an arbitrary address.

3. `relread` (relative read): read an arbitrary number of bytes from an arbitrary offset from the `ngx_http_request_t` struct corresponding to the request.

This model mimics two possible scenarios. First, it models a condition where the client can exploit one or more memory safety vulnerabilities to obtain repeatable arbitrary memory reads and writes. It also models a condition where the client can achieve Turing-complete computation from a single exploit, as if through DOP [40]. The model abstracts and unifies these two exploit scenarios and greatly simplifies exploit development for the CFI bypass proof-of-concept.

We used nginx version 1.19.5 on a server running on Debian 10 (buster), with default runtime protections including full address-space layout randomization (ASLR) with position-independent executable and W^X. The exploit was executed from a separate laptop sending requests over a private local network.

4.2 The Attack

The attack exploits weak memory protections in Linux to overwrite non-writable memory. We target non-writable executable code pages to execute arbitrary code. The attack uses `proc/self/mem`, a file in the `procfs` pseudofile system that exposes access to the virtual address space of a process through a file-like interface. Critically, accesses to memory through this interface **ignore** page read/write/execute permissions.

4.3 File I/O Gadgets

The `proc/mem` attack requires that the attacker can write to a controlled offset of a particular file path through a "file-offset" data-oriented gadget. This requirement is a higher bar than a simple arbitrary file write, which would only write to the beginning of a file. Here, the attacker must control all of the following:

1. The file path;

2. The data to be written;

3. And, the offset to write to.

At first this requirement may not seem realistic; however, we find that both the ubiquitous `stdio` library of libc and the nginx filesystem abstraction layer (which is implemented independently from libc) use data structures that track file offsets. An attacker can modify the fields of these data structures to direct corresponding writes to a desired offset.

Furthermore, the attacker does *not* require a single code path that both opens a controlled file path and performs the file-offset write. Instead, they can use two distinct data-oriented gadgets: one to open the desired path, and one to write at an offset. As long as the attacker-controlled path remains open, the attacker can substitute that file descriptor number into the file-offset write operation.

4.3.1 File-Offset Gadgets

First, the attacker needs a data-oriented gadget to write to a specific offset of a file. Generally this can be accomplished where the application or library tracks the file offset independently of the operating system, and contains code that transparently adjusts the offset before writing. We found that this is the case in both nginx's custom filesystem code and libc's `stdio`. Each used data structures that were expressive enough to permit data-only attacks controlling both the file descriptor and offset of a write.

4.3.2 File Open Gadgets

The attacker must also be able to trigger the opening of a specific file path. Once that file is open for writing, its descriptor can be substituted freely in other data structures used for file I/O in the process, even code that would never use a descriptor originating from targeted `open` call.

4.4 Proof-of-Concept

To demonstrate the attack, we created a proof-of-concept exploit for a *simulated* vulnerability in the open-source nginx web server [10]. We modified a local copy of the source code to intentionally introduce a weakness simulating a vulnerability. These modifications are detailed in Section 4.1.1.

The following are the high-level steps to the exploit:

1. Obtain pointer leaks to infer the randomized virtual memory mappings of the nginx binary, stack, heap, and libc

2. Insert the path "`/proc/self/mem`" to a list of open log files, and use a data-only attack to induce a re-opening of all log files. This creates a valid file descriptor referring to this file, but does not write to it.

3. Modify the in-memory data structures representing the server config to insert an additional proxy route to an attacker-controlled server.

4. Send a request to the newly-inserted proxy route. The attacker-controlled server artificially slows its response over an interval of several seconds.

5. While the server response is pending, the attacker initiates additional requests to leak heap memory until it locates the data structures corresponding to the pending slow response.

6. Modify the response data structures of the slow response as if the response were using temporary file buffering, using a crafted object with the file descriptor set to the open `/proc/self/mem`, the offset set to the virtual address of the nginx code, and the buffer pointing to a payload to execute.

4.4.1 Arbitrary File Open

The first step is to use a data-only attack to open a file path of the attacker's choosing and hold the file descriptor open. We will later use this descriptor in the file-offset write attack.

This principle applies not only to this particular proof-of-concept, but also in general: the attacker can use one code path to first open a file and another to write to it. Thus, the attack does not need a single data-oriented gadget that is expressive enough to meet all the requirements at once.

In the case of nginx, a global variable was used to trigger reopening of log files. The log files were also stored in a global list data structure. We were able to modify the list of log files to insert an entry to `proc/self/mem`. Then, writing a 1 to the `ngx_reopen` variable caused the process to open our path. Listing 4.1 shows a sample of the relevant code.

4.4.2 File-Offset Write

Nginx is unusual in that it avoids using a standard C file library in favor of its own filesystem abstraction implementation. However, like libc, its file-related data structures implement offset tracking with transparent seeking. These data structures are therefore an appropriate target for a file-offset write data-oriented gadget.

Listing 4.2 shows an interesting subset of the data structures nginx uses to represent files and temporary files. Nginx uses these structures for most file I/O tasks, with the notable exception of log files. One feature that *does* use these structures is the use of temporary files for proxied response buffering via the `proxy_temp_path` config directive [9]. Our exploit makes use of the code path for this feature.

Listing 4.3 further shows how this data field is used in writing to temporary files. The function `ngx_write_chain_to_temp_file` takes the value from the `offset` field and uses it as the offset for the write. Furthermore, the `ngx_event_pipe_t` structure encapsulates both the temporary file structure *and* the buffer structure that determines the file's contents. As a result, control of the single `ngx_event_pipe_t` object on this code path permits a data-only attack to control the file descriptor, contents, and offset of the file write.

Note that the attacker also must control the file path, not only the descriptor. While this same data structure does carry file path information, the code uses this field to *create* a temporary file, refusing to open it if it already exists. Because the `proc/mem` attack targets an existing file path, the `path` field cannot be used. Instead, we use an independent data-only attack to obtain a valid file descriptor for `proc/self/mem` as detailed in Section 4.4.1.

```
        if (ngx_reopen) {
            ngx_reopen = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs")
    ;
            ngx_reopen_files(cycle, -1);
        }
```

Listing 4.1: A global variable can cause log files to be re-opened, allowing the attacker to obtain a file descriptor for an arbitrary path.

```
typedef struct ngx_file_s {
    ngx_fd_t                   fd;
    // ...
    off_t                      offset;
} ngx_file_t;

typedef struct {
    ngx_file_t                 file;
    off_t                      offset;
    ngx_path_t                *path;
    // ...
} ngx_temp_file_t;
```

Listing 4.2: Data structures used by nginx to represent files. These structures encapsulate the three elements needed for the data-only attack: the path file descriptor and offset for file writes.

```
ssize_t
ngx_write_chain_to_temp_file(ngx_temp_file_t *tf, ngx_chain_t *chain)
{
    // ...
    return ngx_write_chain_to_file(&tf->file, chain, tf->offset, tf->pool)
    ;
}
```

Listing 4.3: Code snippet showing how nginx uses the file descriptor and offset from the data structure for writing.

4.4.3 Creating a Favorable Configuration

Having noted `ngx_event_pipe_t` as a worthwhile target, the attacker must determine a code path using this structure that is reachable in the course of legitimate control flow. By examining the function call graph and source code, we identified the response buffering feature as an interesting target. This feature allows buffering of large request and response to temporary files on disk, and it is highly configurable for options like file path, size, and more [9]. It also is only used for proxy routes — URLs configured to serve data from an upstream server.

Ideally, our attack should not rely on any particular pre-existing configuration. Instead we can use a data-only attack to modify the in-memory data structures that represent the configuration and manipulate the configured behavior of nginx. This is relatively straightforward because this configuration data is stored in writable memory, as is typical for dynamic configuration data parsed from a config file. We used an iterative process to construct configuration data structures and insert them into process memory (other research has suggested that this process can be automated, potentially reducing barriers to exploitation [40]). The iterative process involved inserting minimal copies of configuration structures into memory under a debugger, breaking when an error occurred, and revising the data structure until the desired behavior was achieved. Using this method we were able to insert data structures representing a new proxy route in a server that previously had none configured.

This case study suggests that data-only attacks can configure almost any behavior the application is capable of under normal execution. Hence, attackers may have access to greatly expanded attack surface by modifying runtime configuration in order to induce whatever state facilitates further exploitation. In other words, data-only attacks can be chained together to reach data-oriented gadgets not otherwise accessible.

4.4.4 Extending Object Lifetime

The last remaining challenge for completing the proof-of-concept is to actually locate and overwrite the `ngx_event_pipe_t` structure in-memory after it is created but before it is used.

The typical lifetime of an `ngx_event_pipe_t` object makes this non-trivial: an object is created to handle the request and destroyed when the request is completed. It may be possible to locate and overwrite this object during the lifetime of a single request using a data-oriented programming attack. Such an attack could find the object in memory and make the necessary changes using Turing-complete computation [40]. However in this case we found that it was possible to proceed without resorting to this technique.

Instead we artificially extended the lifetime of an `ngx_event_pipe_t` object by taking advantage of high-level application behavior. In Section 4.4.3 we described inserting a proxy route to expose the code path containing the file-offset write gadget. We can use the same proxy route to extend the lifetime of an `ngx_event_pipe_t` object. By setting the proxy route to an attacker-controlled server, the attacker can artificially delay responses to create long-lived response objects in memory. The attacker sends a request to the newly-configured and artificially slowed proxy route. Concurrent requests sent by the attacker then have a comfortable window to locate and modify the corresponding `ngx_event_pipe_t` object in memory.

4.4.5 Finalizing the Attack

With the slow request pending, we send concurrent requests repeatedly leaking parts of heap memory until the `ngx_event_pipe_t` object is found. We then overwrite it with a crafted object containing the file descriptor for `proc/self/mem`, an offset corresponding to the virtual address of the non-writable executable code segment, and a buffer containing the machine code payload. Nginx uses these fields to perform a write to the `proc/self/mem` file at the specified offset. This interface ignores page read/write/execute permissions and copies the payload directly into unwritable executable memory. When the syscall completes, control is returned to the attacker-controlled machine code.

4.5 Discussion

This proof of concept suggests that the `proc/mem` is feasible, but aspects of our attack depended on the specific structure of Nginx's file I/O layer. To some extent, all data-oriented attacks are situational since they rely on the particular operations available in the application's control-flow graph; however, file writing data-oriented gadgets are common. Data-oriented attacks targeting file writes have been show to be possible in a variety of common software before [22], and the `proc/mem` technique can convert these into arbitrary code execution as long as the attacker can control the file offset.

Still, the question remains how common not only are file *write* gadgets, but file *offset-write* gadgets. To help answer this question we examine the most common file I/O library for C applications on Linux, the GNU standard C library `glibc`.

4.5.1 File-Offset Data Gadgets in `glibc`

It turns out that the ubiquitous GNU standard C library contains a gadget very similar to the one exploited in Nginx. Listing 4.4 shows part of the code that constitutes the file-offset write gadget. When `glibc` flushes a file's buffers to disk, it checks and possibly calls `seek` before `write`. Figure 4.1 shows the data flow from the `FILE*` data structure to the parameters of both calls. Hence, all of the necessary parameters for the `proc/mem` technique are attacker-controlled in a data-only attack. Furthermore, `glibc` *automatically* flushes all open `FILE*`s when they are closed or the application exits, so the attacker may not even need to be able to reach a code path that actually performs a file write in the course of normal execution. In other words, there are only two prerequisites to performing the `proc/mem` attack with `glibc`:

1. The attacker must be able to persuade the application to obtain and keep a file descriptor for `proc/self/mem`;

2. The attacker must be able to overwrite a `FILE*` structure.

```
1
2  if (fp->_IO_read_end != fp->_IO_write_base)
3      {
4        off64_t new_pos
5      = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
6        if (new_pos == _IO_pos_BAD)
7      return 0;
8        fp->_offset = new_pos;
9      }
10   count = _IO_SYSWRITE (fp, data, to_do);
```

Listing 4.4: A snippet of code from `glibc` showing how file buffers are synced to the file system allowing the attacker to control the offset.

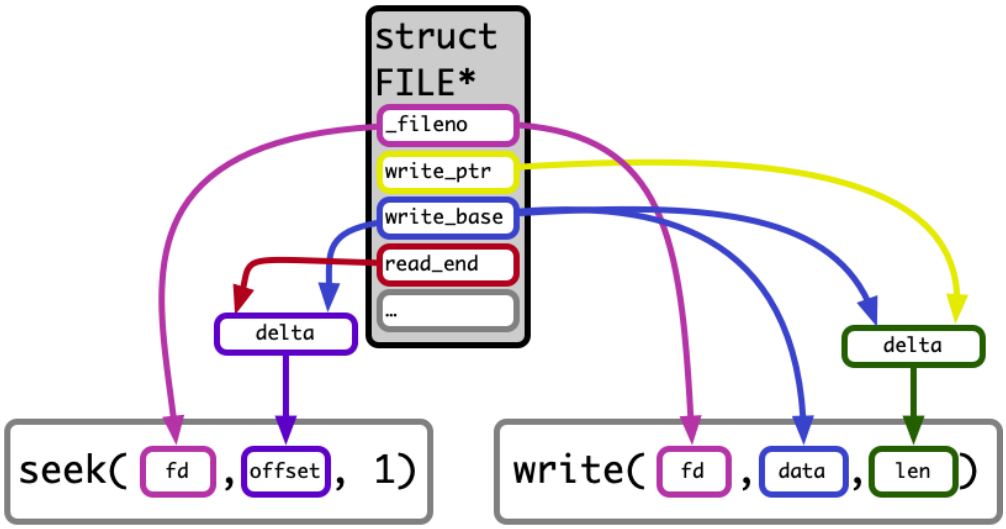

Figure 4.1: A data dependency graph illustrating how all parameters to a `seek()` and `write()` call are controlled by a single data structure in the standard C library, constituting a file-offset write data gadget.

We believe these two criteria are frequently met in real-world memory corruption vulnerabilities. As future work, we propose a survey of common software using past vulnerabilities to obtain a better estimate for how widely-applicable the `proc/mem` attack is.

Chapter 5

Strengthening Executable Memory Protections

In the previous sections we showed how weak memory protections in Linux can violate key assumptions in defense systems. We now set out to mitigate these gaps. Not only do we want to block the known vectors for tampering with executable memory, we also must minimize attack surface and systematically validate the patches to achieve the highest possible confidence that the system is secure. Our goal is to engineer a system that brings real-world code into as close alignment as possible with W^X assumptions. Accordingly, we want to prevent *all* unauthorized changes to both contents and layout of executable memory after program initialization is complete, including both direct changes via memory accesses and indirect changes via the kernel. As a secondary goal, we want to allow some explicitly authorized and specifically defined changes to executable memory in order to support applications that require it, such as JIT compilers.

First, we introduce a new syscall `xlock` that permanently "locks" the executable address space and memory contents of the calling process. The purpose of this syscall is to make our changes to the memory permissions model "opt-in." That is, each individual process must explicitly request the enhanced code integrity enforcement policy. Thus, security-sensitive applications, such as services exposed to the internet, can benefit from increased protection without threatening system-wide stability in the presence of other applications that may not be compatible with the xlock model.

Finally, we validate our implementation using custom modifications to PeX, an automatic Linux permissions checking framework [81].

5.1 Goals

The goal of the `xlock` system is to provide a robust and comprehensive guarantee of executable code integrity for the lifetime of a process. `xlock` can thus be used either on its own to prevent code injection as part of an exploit chain, or as a foundation for other systems rely on code immutability as part of the W^X assumption.

5.2 Design

The `xlock` syscall is designed to lock all executable memory in a process. When a process wants to prevent further modifications to its code, it may call the `xlock` syscall and check the return value to ensure the call succeeds. After the call is made, it sets a flag in the kernel data structure associated with the process, marking the process as xlocked. This flag cannot be unset from userspace. The design of the `xlock` system includes a setup phase that pins all executable pages in memory, added permission checks for memory-related operations, and optional configurable exceptions for loading vetted dynamic libraries or JIT pages.

5.2.1 Setup

The setup phase executes immediately in the kernel when the `xlock` syscall is invoked. This phase traverses the virtual memory mapping of the calling process, pins all executable pages in memory, and detaches them from any backings such as files or shared memory. In the language of the Linux kernel, this effectively converts all executable pages into private, anonymous mappings. If a page is not yet loaded, this phase allocates physical memory for it and copies it from disk. The mapping is then updated to remove its reference to the backing, preventing changes from propagating to the physical memory.

On its own, this step effectively prevents several of the attacks detailed in Section 3.2. Many of these attacks exploited executable memory mappings that were backed by

a mutable resource such as a file. Although the memory mapping itself was non-writable, an attacker could circumvent memory mapping permissions by modifying the underlying resource directly. In many cases, the kernel would then propagate these changes to physical memory. We rule out this class of vulnerability by moving all executable pages to physical memory and unlinking them from any other backing.

5.2.2 Permission Checks

Setting the `xlock` flag also enables a set of enhanced permission checks for certain memory-related syscalls. Any operation that causes a change to the process's executable code should be denied. These checks must be added to syscalls that have a straightforward ability to change executable code like `mmap` and `mprotect`, but also to interfaces with more obscure code-modifying properties like `mremap`, `proc/self/mem`, and `ptrace`. It is a major challenge to locate and add checks to every possible kernel code path that could modify memory mappings or contents from userspace. Section 5.4 details our strategy for addressing this challenge.

5.2.3 Dynamic Code

Some applications actually require changes to their executable code at runtime. Examples include language interpreters, VMs, or emulators that use just-in-time (JIT) compiling to produce executable machine code at runtime as well as applications supporting extensions or plugins in the form of dynamic libraries. Ideally the `xlock` system would retain support for these applications without allowing unauthorized modifications to code. The `xlock` syscall allows the process to designate an external userspace process to act as the "gatekeeper" and dictate access to new executable code.

The gatekeeper process communicates with the kernel using the `ptrace` interface and runs with elevated privileges to protect it from tampering from the application. Whenever the application maps a new range of executable pages using `mmap`, `mprotect`, or similar, the kernel first maps these pages into the gatekeeper process as writable pages and notifies it of

the new virtual address range. The gatekeeper is then free to read and modify the newly-mapped pages (as well as the rest of the application's memory), performing any desired validation or sanitization. The gatekeeper then notifies the kernel to allow or disallow the mapping change.

This architecture is flexible and allows any type of validation. For example, this design could be used to enforce digital code signing. The gatekeeper can require that any executable pages loaded dynamically have a valid digital signature from a trusted source.

5.3 Implementation

We implemented a subset of this design as a set of kernel patches, working on the Linux kernel version 5.4.24. The implementation adds support for the `xlock` syscall including the setup phase and set of new permission checks, but not the gatekeeper architecture.

The `xlock` syscall implementation traverses the virtual memory mapping of the calling process and performs the following procedure for each executable mapping:

1. Ensures the pages are faulted into physical memory;
2. Allocates new physical memory and a new page table entries (`pte`) mapping to the new pages;
3. Copies the memory contents of the mapping to the new pages;
4. Marks the `pte`(s) as readable and executable only;
5. Invalidates the page range in the MMU;
6. Closes any files associated with the `vm_area_struct` for the mapping;
7. Removes any backings associated with the `vm_area_struct` and marks it as anonymous;
8. Updates the permissions on the `vm_area_struct`, making it readable and executable only.

This procedure converts all executable mappings in the process to *private* and *anonymous* mappings. Copying the contents of the mapping to new physical memory disassociates the mapping from any shared memory, ensuring that the mapping is private (i.e., not shared with any other mapping). Removing and closing any files associated with the mapping also makes it anonymous (i.e., not associated with any file or other resource). Private, anonymous mappings are thus immune to the entire class of memory-permission bypasses that depend on inconsistent permissions for multiple mappings or associated backings because they are not associated with any other resource or mapping.

The `xlock` syscall also irrevocably sets a flag in the `task_struct` of the process that marks it as executable-locked. When this flag is set, subsequent requests from userspace to modify virtual memory mappings associated with executable code are denied. Our patch adds checks to system calls such as `mprotect`, `mmap`, `mremap`, and more that are capable or creating, moving, modifying, or deleting executable mappings as well as interfaces that can bypass page protections such as the `proc/self/mem` pseudo-file. While the logic of the checks is very straightfoward, discovering all such kernel code that may be reachable from userspace is non-trivial. We detail our process for identifying locations that need new checks as well as a comprehensive listing of added checks in the next section.

5.3.1 Userspace Support

In addition to kernel patches, we also wrote a small userspace library that adds an interface for C/C++ and Python applications to the `xlock` syscall. This library allows applications to explicitly call `xlock` after the dynamic linker finishes loading executable and library code.

We also developed an `xlock` shim command line utility that allows existing applications to be executed with the strengthened `xlock` memory model without changes to the application. The shim uses Linux's dynamic linker to inject the shim library into the application at run time. The library calls `xlock` after the application and library code is finished loading, but before control transfers to the application's `main` function. This tool is particularly useful for adding drop-in support to binary applications where the user may not have access to source code.

5.4 Validation

As we argued in Section 3.3, interjecting a new security boundary into an existing and sprawling code base is an error-prone process [81]. Developers have written millions of lines of code and hundreds of kernel interfaces with a more permissive security model in mind. Any of these interfaces may require new checks to enforce the new boundary. Relying on a manual search of documentation and code carries a high risk of overlooking some functionality that violates the desired security property. Of course, even a single overlooked check can render all others moot. To address this challenge, we use static analysis to automatically discover code paths reachable from userspace with the potential to modify userspace memory contents or mappings. This approach is not totally fool-proof, but it does greatly reduce the manual burden of discovering

We validated our changes and identified unchecked code paths using a modified version of PeX, an automatic permissions checking framework for the Linux kernel by Zhang et al [81]. PeX uses the LLVM [13, 49] intermediate representation of the Linux kernel to construct a call graph of the whole kernel image, resolving indirect calls using a heuristic called KIRIN. The call-graph is then used to construct a more detailed interprocedural control-flow graph, which links *intra*-procedure control flow graphs with edges between function calls and the entry and exit points of possible callees. PeX then uses a set of known permission-check functions to infer privileged functions and employs dominator analysis to identify privileged functions that can be reached without a permission check. Finally, the tool outputs a list of call stacks that reach the identified critical functions, both *good* (code paths with a permission check) and *bad* (code paths missing a permission check).

While PeX supports *automatically* inferring privileged functions, for our purposes this functionality results in unnecessary noise from unrelated code. We are instead focused on a small set of particular functions that may manipulate memory contents or mappings. For this reason, we modified a copy of the PeX framework to instead accept a configured list of critical functions. By inspecting existing kernel code for functions like `mmap` and `ptrace`, we formulated a list of internal utility functions used by the kernel for 1) modifying memory protections of userspace mappings and 2) accessing userspace memory without enforcing

memory protections. These functions are listed in Table 5.1. Likewise, we limited the set of permission checks recognized by PeX to only include `mm_is_xlocked`, a utility function added by our patch that checks if a process' address space is executable-locked.

We first added checks at 6 known locations that could violate W^X, using the attacks detailed in Section 3.2. These checks served as a baseline to validate that PeX was indeed able to identify the known cases as having *good* permission checks. In addition to correctly identifying the 6 good permission checks, PeX also found 7 unchecked paths reaching critical functions. Table 5.2 lists all checks added, both through our initial manual analysis and through PeX output. Notably, PeX found 3 *legacy* system calls that are no longer in normal use, but may still exist on some systems. It also found surprising functions such as `madvise` that only violate the "executable-locked" invariant as a result of particular flags or options, and not as the typical usage of the syscall.

5.5 Performance Impact

The performance affects of `xlock` can be broadly divided into three categories:

1. The one-time overhead of calling `xlock` after initialization.
2. The ongoing run-time overhead after calling `xlock`.
3. The effect on memory usage after calling `xlock`.

The one-time cost of calling the `xlock` syscall is small but measurable, and depends on the size of the executable memory mapped by the process. We measure and report the setup time for variously sized executables in Figure 5.1. For a process with 4.9MB of mapped executable memory, the `xlock` syscall took approximately 2ms. Since this cost is only incurred once during the program initialization, it is likely to be imperceptible to users.

The `xlock` system does not incur substantial run-time overhead in applications that do not frequently map new executable code after calling `xlock`. The potential source of run-time overhead is in the added permission checks (listed in Table 5.2), which must run

Table 5.1: Critical functions used for discovering unchecked paths that could potentially modify executable memory or mappings.

Function	Description
`vma_set_page_prot`	Modifies page protections.
`get_user_pages_remote`	Prepares to access userspace memory, explicitly ignoring page protections.
`__get_user_pages_locked`	Identical to `get_user_pages_remote` when called with the `FOLL_REMOTE` flag.

Table 5.2: All kernel functions where `mm_is_xlocked` checks were added.

Function	Found	Description
`__ptrace_may_access`	Manual	Internal permission check governing variety of operations on processes.
`do_mprotect_pkey`	Manual	Changes memory mapping permissions.
`do_mmap`	Manual	Maps new memory.
`sys_ptrace`	Manual	`ptrace` syscall entrypoint.
`mm_access`	Manual	Internal check for access to process's virtual address space.
`sys_mremap`	Manual	Syscall that may move mappings.
`sys_uselib`	PeX	Deprecated syscall for loading libraries.
`__do_execve_file`	PeX	Used to execute new programs.
`__do_munmap`	PeX	Unmaps memory.
`sys_remap_file_pages`	PeX	Deprecated syscall that can move mappings.
`do_brk_flags`	PeX	Legacy memory management syscall.
`apply_vma_lock_flags`	PeX	Internal helper function used by `mlock` family that may change mapping permissions.
`sys_madvise`	PeX	Syscall that may unmap pages.

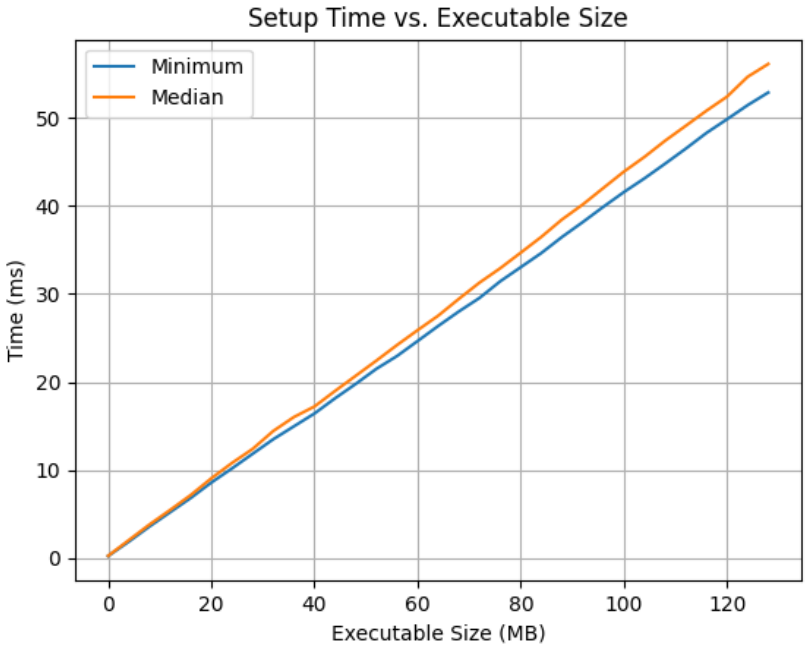

Figure 5.1: Benchmark for cost of calling `xlock`, as a minimum and median of 10,000 runs each for various sized executables. There is a linear relationship between executable size and setup time, which corresponds to the time required to reallocate private, anonymous memory and possibly read from disk for each executable page. This cost is only incurred once, at application initialization.

for all userspace applications (even ones that do not user `xlock`). The added permission checks, when inlined, amount to a single additional memory access and conditional branch. Furthermore, since the checks all reside in code paths that modify virtual memory mappings or bypass page protections, none of them are called frequently in typical workloads. Overall, the effect of the added checks in rarely-used code results in no measurable performance penalty at run-time.

In terms of memory use, `xlock` requires pinning all executable pages in main memory. While this does not increase the *virtual* memory usage of a process, it does put additional pressure on system *physical* memory by precluding use of demand paging. We calculate an estimate for increased physical memory usage by measuring the number of executable pages not resident in physical memory in during a sample execution of a program. Our results are listed in Table 5.3. Exact numbers for demand paging systems will depend on a particular execution path, as only pages actually executed become resident. Still, these measurements indicate that the extra physical memory usage is generally on the order of a few megabytes, which is acceptable in the context of modern server or desktop systems with several gigabytes of physical memory.

5.6 Comparison to SELinux

SELinux provides controls that can prevent some methods for injecting executable code at runtime, but they stop short of ensuring total code integrity [12]. Table 5.4 lists these checks and the operations that they apply to. These restrictions can enforce the W^X policy on *page permissions*, ensuring that no mapping that is or has been written to can also be executed. However, as we have seen, this does not translate to a guarantee of code integrity. Executable-mapped files can be modified without ever mapping them writable. Shared memory mappings can refer to the same physical memory with different permissions. The kernel provides interfaces that can modify memory contents regardless of mapping permissions, such as `ptrace`—an operation that bypasses *all* existing checks, including those of SELinux, when the thread group of the target matches that of the caller. In short, SELinux can ensure that

Table 5.3: Estimate for resident memory increase incurred by `xlock` in common server software.

Software	Memory Increase
OpenSSH (7.9p1)	2.4 MB
Python (3.7.3)	1.0 MB
Nginx worker (1.19.5)	1.1 MB
JVM (OpenJDK 11.0.9)	9.6 MB

Table 5.4: Executable-memory related controls provided by SELinux, and the corresponding operation they allow or deny.

Permission	Operation
`execmod`	Mark page executable that was previously writable.
`execmem`	Map a page both executable and writable.
`execstack`	Map any part of the stack executable.
`execheap`	Map any part of the heap executable.

memory mappings adhere to the basic form of the WˆX policy, but cannot guarantee code immutability.

5.7 Discussion

The `xlock` system systemically strengthens the WˆX assumption on Linux. It fixes an important subset of the known attacks on PKU-based intra-process isolation architectures and prevents the `proc/mem` attack as a bypass for control-flow integrity. The WˆX assumption is fundamental to the design of both systems, so gaps in its implementation on Linux can undermine the security of each. A CFI system can call `xlock` as part of its initialization, thereby preventing the `proc/mem` attack *and* other potentially unknown violations of WˆX in the protected application or via the kernel. By copying all executable memory into private, anonymous mappings, `xlock` excludes an entire class of known attacks that rely permission inconsistencies between mappings and their backing resources. The use of static analysis to automatically detect missing checks further bolsters confidence that the new `xlock` permission checks are comprehensive.

While our work focuses on userspace memory mappings, previous work has uncovered WˆX violations and exploited them in the context of kernel memory mappings. Liakh et al. used a formal model to detect violations of WˆX in kernel memory permissions [50]. Kemerlis et al. later demonstrated the "ret2dir" attack, which is conceptually similar in that it exploits dually-mapped pages with inconsistent permissions to bypass Supervisor Mode Access Prevention (SMAP) in kernel exploits [44].

Chapter 6

Improved Architecture for PKU-based Isolation

Using the lessons learned from previous work, we propose an improved architecture for intra-process isolation that remains efficient but better facilitates secure implementation. In Section 3 we described novel weaknesses that we found in two similar PKU-based intra-process isolation systems, although they were each designed and implemented independently of the other. We analyzed the root cause of these weaknesses and argued that they stem from similar design choices. As a class, the vulnerabilities shared a common cause: the kernel, unaware of intra-process boundaries and authorized to access all process resources, could be manipulated into violating the intra-process security model. Although each of these loopholes could be individually fixed by changing some implementation details, preventing the *class* of vulnerabilities requires a systemic solution.

We propose an architectural change that would fundamentally modify the way the kernel views intra-process components in order to mitigate this class of attacks. Our change allows the kernel to treat intra-process components as separate process for most purposes, without requiring any explicit interaction with, or additional context switches to, the kernel. In this architecture each intra-process component is associated with a "virtual" process which we call a *coprocess*, spawned by the *primary* process. Coprocesses are never scheduled and cannot run independently of the primary, but they encapsulate a set of process resources that is almost totally independent of the primary.

Our design retains essentially the same model for intra-process isolation from a userspace perspective: the application can designate certain portions of its code to act as the trusted component, and allocate memory that is accessible only from the trusted component. The untrusted component can enter the trusted components only through validated call gates consisting of a `wrpkru` instruction followed immediately by a jump to the trusted code. Unsafe `wrpkru` gadgets can be neutralized either by static rewriting (as in ERIM) or dynamic hardware breakpoints (as in Hodor).

However, the coprocess architecture substantially changes the kernel model for intra-process isolation. As the `seccomp` filter exploit in Section 3.2.5 illustrated, memory cannot be securely isolated without also isolating other intra-process resources. Otherwise, it is possible for the untrusted component to indirectly and surreptitiously influence behavior of other components. In order to isolate non-memory resources and mitigate kernel confused-deputy attacks, we model intra-process components as *separate processes* to the kernel for most purposes. We also integrate the `xlock` system described in Section 5 in order to preserve the required static invariant of application code that no exploitable `wrpkru` exists.

6.1 Background and Goals

In general, the purpose of intra-process isolation is to mitigate the impact of an application or library vulnerability by efficiently separating in-process components. A single bug can allow attackers to cause the impacted component to behave arbitrarily [66]. In a traditional non-isolated process, such an exploit potentially exposes any resource accessible to the entire process, which frequently includes data that is irrelevant to the vulnerable component [29]. Many works describe methods for limiting access between intra-process components without resorting to the overhead required for a full context switch between the kernel or another process [25, 52, 46, 53, 74, 77, 23, 32, 59]. The fastest, ERIM and Hodor, use specialized hardware known as Protection Keys for Userspace (PKU) [38, 72].

Our contribution refines these two systems by accounting for practical concerns with secure development, using lessons learned in Section 3. We outline architectural modifications to this scheme that help facilitate secure implementation. In other words, our goal

is to minimize the opportunity for implementation weaknesses caused by human error both during initial development and subsequent maintenance.

Of course, we also endeavor to limit overhead to an acceptable level; intra-process isolation is useless if it doesn't outperform traditional context switching.

6.1.1 Component Model

For simplicity, our model for intra-process components primarily focuses on dividing the process into a trusted and untrusted component, defining privileged blocks of code which together form the trusted component. These blocks of code exist within functions, not as self-contained and separate callable functions. Components are orthogonal to scheduling. Threads enter and leave the trusted component depending on the code they are executing at the moment.

This model simplifies certain design considerations, but it can also be easily expanded to accommodate multiple independent components or modified to correlate components with other entities such as threads, functions, or libraries.

6.1.2 Threat Model

We continue to operate under a similar threat model used in comparable works [72, 38], with a small change. We assume the following:

1. The attacker may freely read and write memory in the untrusted component, subject to page table permissions.

2. The attacker can arbitrarily manipulate the control flow of execution in the untrusted component, but cannot inject new executable code.

3. Any unsafe `wrpkru` gadgets have been neutralized either through static rewriting [72, 71], dynamically-inserted hardware breakpoints [38], or a similar technique.

4. The hardware, kernel, and linker will not behave *maliciously*, although they may be agnostic of intra-process boundaries.

5. The application is trusted up until a coprocess is initialized (in practice, this means initializing the coprocess *before* exposing attack surface).

We do not state as an assumption that our implementation is secure, although we will argue why we believe it is *more likely* to be secure compared to other designs.

6.2 Design and Prototype

The design encompasses the following:

1. New kernel code and data structures to dynamically look up and use the isolated coprocess structure as a proxy for the ordinary process control block in most operations;
2. New options for the `clone` syscall for informing the kernel of new isolated components by spawning a new coproc;
3. A statically-linked library providing userspace support for creating coprocesses, swapping between them, and allocating isolated memory.
4. The `xlock` system, underlying the assumption of executable code immutability.

6.2.1 Kernel Internals

We model intra-process components as *coprocesses*, which appear as independent child processes in the process tree of the *primary* process that created them. Each coprocess has an associated PKRU value that grants it access to isolated memory, as well as a nearly complete independent set of process attributes from the kernel's perspective.

Process Control Block

In the Linux kernel, each process is represented by a C structure known as `task_struct`. The coprocess system allocates an additional `task_struct` for each coprocess and dynamically substitutes the appropriate one while in kernel space. This structure holds (either directly or by reference) all information about a process such as PID, user and group IDs, memory

mappings, file descriptor tables, and more. The definition for this massive structure spans almost 700 lines of code and includes approximately 250 fields (although the exact number varies depending on the build configuration).

The coprocess design assigns each component its own `task_struct` in the kernel. A primary process maintains an array of its coprocesses as pointers to `task_struct` structures, indexed by the protection key associated with that component. Coprocesses likewise keep a pointer to their primary. This data structure is illustrated in Figure 6.1. Coprocess component swaps remain entirely in userspace and primarily mediated through the PKRU. When a coprocess or primary *does* enter the kernel for some reason (syscall, interrupt, etc.) the kernel reads the PKRU value to determine which coprocess, if any, is active. Figure 6.2 visualizes a simple example of two different intra-process components each making a call to `getpid`.

This design causes the in-process components to *automatically* inherit most of the existing **inter**-process security boundary enforcement of the kernel. For example, file I/O system calls issued from one component cannot possibly access or change the file descriptors opened for another component, absent a bug in the existing security model of the kernel. This property is achieved only by virtue of isolating the `task_struct` structures, without any changes to kernel file I/O code. The same principle applies to other process resources and metadata such as `seccomp` filters, namespaces, and more.

However, some fields of the `task_struct` remain shared between intra-process components: namely, the scheduling, memory map, and CPU context. We detail the changes needed to facilitate these shared resources in Subsection 6.2.4.

The `current` Process

The kernel source code refers to the `task_struct` of the running process through the `current` macro, so by modifying this macro we can substitute the `task_struct` of the current coprocess for the `task_struct` of the primary. On x86, the `current` macro is a synonym for a trivial function which returns a per-CPU global variable, shown in Listing 6.1. The global variable is nothing more than an ordinary pointer to a `task_struct` that is set in the kernel in the course of scheduling a new process on a core. Subsequent traps to the kernel via

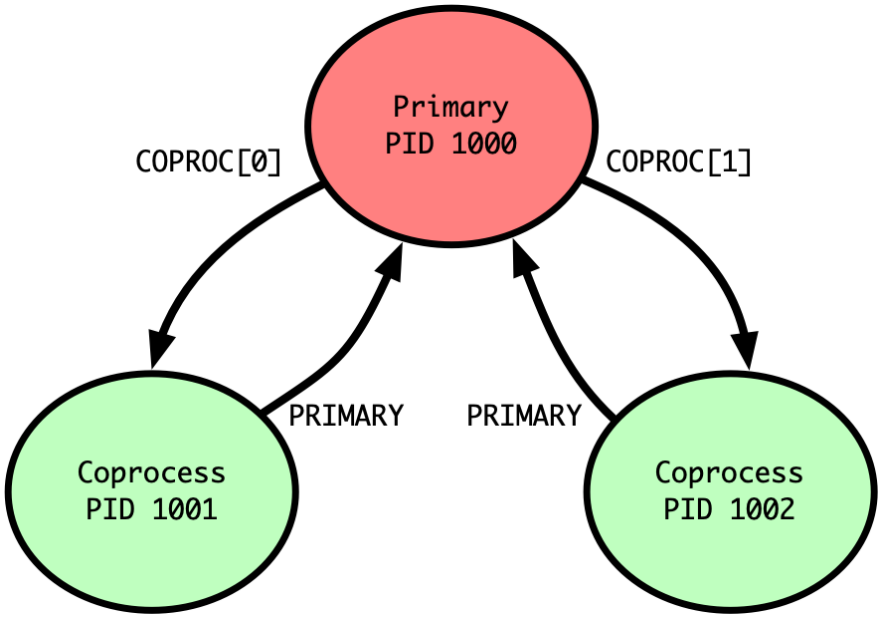

Figure 6.1: Each process maintains an array associating protection keys to coprocesses, and each coprocess maintains a reference to its primary.

```
static __always_inline struct task_struct *get_current(void)
{
  return this_cpu_read_stable(current_task);
}

#define current get_current()
```

Listing 6.1: The usual definition of the `current` macro simply returns the per-CPU variable representing the task currently running on the CPU.

```
static __always_inline struct task_struct *get_current(void)
{
    struct task_struct *tsk = this_cpu_read_stable(current_task);
    struct task_struct *coproc;

    coproc = get_coproc(tsk, safe_rdpkru());
    if (unlikely(coproc)) return coproc;

    return tsk;
}
```

Listing 6.2: The patched `current` definition potentially returns a different struct depending on the current state of the PKRU register.

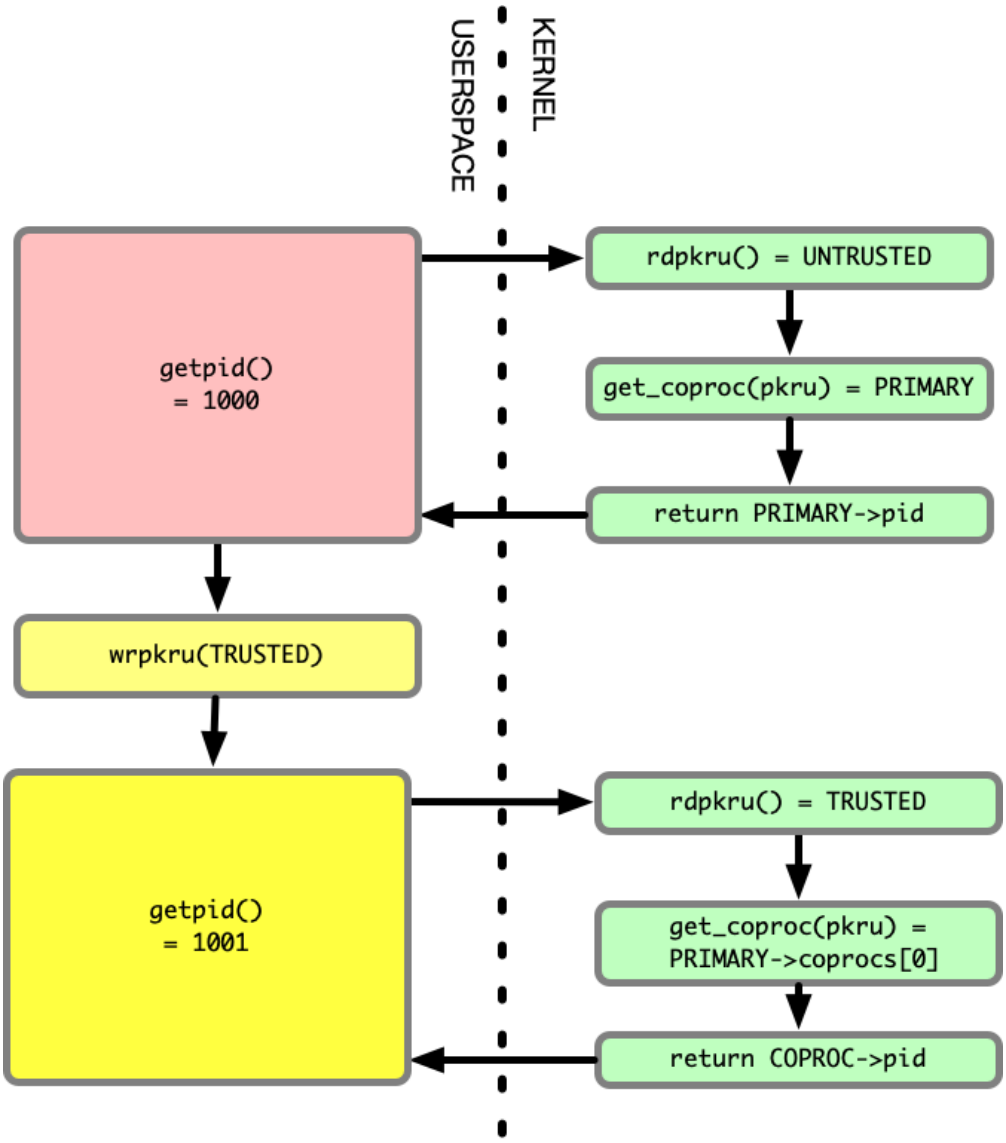

Figure 6.2: Every reference to the currently running process from the kernel uses the current PKRU value to lookup an individual `task_struct` encapsulating isolated process resources for the appropriate coprocess.

syscall, interrupt, or similar do not modify this variable until the kernel explicitly schedules a new process.

A simple change, illustrated in Listing 6.2, causes `current` to find the correct coprocess. Now, the value of PKRU register is used to index a 16-element array of pointers to the `task_structs` of possible coprocesses. This modification transparently causes the kernel to differentiate between intra-process components, segregating shared resources automatically.

This single change has wide-reaching effects, since nearly all kernel code that deals with userspace references the *current* process. Table 6.1 provides our accounting of *current* usages and related references. The macro is referenced over 9000 times in the Linux source code (although the number of those references that are actually reachable depends on the build configuration). Even with a minimal configuration that excludes many common feature and options from compilation, `task_struct` members are accessed many thousands of times in the resulting binary. Changing the `current` macro effectively instruments several thousand accesses to make them aware of intra-process components by default, all at once.

6.2.2 System Interface

Because coprocesses are primarily orchestrated in userspace, the system interface for them is quite small. We modified the `clone` system call [5] to support a new optional flag (`CLONE_COPROC`) indicating that a new coprocess should be created. The `clone` syscall is an overloaded system call that can be used interchangeably with `fork` or can be used to create new threads, so it is natural to extend it to create new coprocesses as well. In response, the system returns both the PID and the protection key (between 1 and 15, inclusive) associated with the new coprocess. However, unlike a conventional `clone` or `fork` operation, the call returns only once, since coprocesses are not schedulable on their own.

Userspace manages switches between components, so no new kernel interface is needed for this purpose.

Table 6.1: Summary of statistics on usage of `current` macro and `task_struct` references in the Linux kernel source code.

Operation	References
`current` (Minimal Config, LLVM)	400
`current` (All Source)	9169
`task_struct` dereferences (Minimal Config, LLVM IR)	7846

6.2.3 Userspace Support

Userspace support for coprocesses comprises a thin static library called `libcoproc` providing convenience wrappers and macros for common functionality. A `coproc` function creates a new coprocess and returns the PID and protection key for it. A pair of macros (`coproc_swap_to_trusted` and `coproc_swap_to_untrusted`) expands to secure call gates [72] for component swapping. The library is statically linked to avoid breaking intra-process boundaries by forcing a trusted component to rely on a potentially untrusted global offset table for calling out to `libcoproc`.

6.2.4 Shared Resources and Edge Cases

Most kernel subsystems transparently handle the dynamically-retrieved `current` process without modification, but some require special treatment. Some code must be modified to *explicitly* share certain intra-process resources, such as the virtual address space and FPU context. Others need to adopt slightly different semantics in the context of a coprocesses.

Virtual Address Space. Coprocesses must share a virtual address space, because there is no opportunity to change out page tables during a userspace-only context switch. The PKU hardware instead mediates access to subsets of the address space assigned to the trusted component. Such an arrangement is not unprecedented; threads in Linux are essentially just processes that share a reference to the same virtual address space. Hence, existing kernel code is already savvy to the possibility of shared address space between different processes and no further changes are needed to facilitate the sharing.

Scheduling. Since coprocesses are never scheduled, they necessitate some changes to the Linux scheduling system. The Linux scheduling code is modular and allows individual processes to use different schedulers. Coprocesses are simply assigned a custom scheduler that never queues the coprocess. Instead, when the primary process is scheduled, the last PKRU value is restored which effectively restores the correct coprocess context.

FPU Context. When the kernel switches between processes, it stores the userspace FPU context of the old process (which includes the PKRU value) in a member of the `current`'s `task_struct`. A surprising side effect of the "by-default" isolation of process resources was that the FPU context was stored to the `task_struct` of the current *coprocess*—but restored from that of the *primary* process, since this process is the one that is actually scheduled. This behavior caused the wrong PKRU value to be restored during some context switches between userspace processes. We addressed this bug by explicitly sharing FPU context between coprocesses.

Signals. As a mechanism that interrupts ordinary scheduling of a process and suddenly jumps to different code, signal handling poses some problems for intra-process isolation. Our solution is to redirect all signals sent to coprocesses to their primary, which can handle the signal with an unprivileged PKRU value. Signal delivery can also be delayed while executing in a trusted component to avoid preempting critical code. Signal returns that attempt to assign a privileged PKRU value should be denied.

Exiting. When any coprocess exits, the primary and all other coprocesses also exit.

6.3 Performance

We expect the performance characteristics of coprocesses to be closely related to its predecessors that perform context switching in a similar way, though a new source of additional overhead may lie in the more intensive logic of our modified `current` lookup. With these facts in mind, we replicate some previous measurements of the performance characteristics of PKU itself, and then perform a realistic benchmark of Nginx throughput running with the prototype coprocess system.

6.3.1 Experimental Setup

We ran all of the benchmarks on our server, which has an Intel Xeon Silver 4208 CPU with 8 physical cores running at 2.10GHz. We disabled power optimizations and frequency scaling,

and for microbenchmarks used a technique published by Intel to measure clock cycles used for short sequences of instructions [65]. For the throughput benchmarks, tests were run in a virtual machine running our modified kernel, assigned to 4 physical cores on our machine. The prototype was executed in a containerized environment, and the `ab` (Apache Bench) [7] tool was used to connect to the Nginx instance over a virtual ethernet pair. Software versions used included Nginx 1.20.1 linked to OpenSSL 1.1.1k, running on Debian 10 (Buster) with Linux kernel version 5.4.24.

6.3.2 Microbenchmarks

We measured the cost, in clock cycles, of PKU-based "context switch" compared to a kernel context switch. The actual cost of a context switch depends on a number of application-specific factors such as the state of the CPU pipeline, caching of contiguous code, and possible TLB misses, but these numbers provide a rough idea of the baseline cost of executing particular instructions. We compare a gated direct function call with a trivial system call, `getpid`. A gated direct function call consists of a direct `call` instruction to a trivial function, instrumented with additional instructions to change the PKRU value before and after the function call. The function itself simply returns a value from memory. The result given in Table 6.2 confirm previous measurements [72], which indicated that a gated direct function call is more than twice as fast as a trivial system call.

6.3.3 Throughput Measurement

We integrated our prototype with Nginx [10] and OpenSSL [11] to protect long-term private keys (i.e., RSA keys) in application memory and measured the effect on web server performance. Our experiment was designed to be comparable to the same measurements taken for ERIM [72].

Instrumentation

Nginx was statically linked to `libcoproc` and modified to initialize a new coprocess to act as the trusted component for each Nginx worker. We then linked this version of Nginx

Table 6.2: Microbenchmarks comparing the costs of different forms of intra-process component switches. A trivial system call is approximately three times as slow as a gated direct function call (a `wrpkru` followed by a trivial function call followed by another `wrpkru`).

Operation	Cost (Cycles)
`wrpkru`	20
Gated direct function call	42
`getpid()`	126

with a modified OpenSSL library that switched to the trusted coprocess before each RSA decryption, and switched back to the untrusted context afterwards.

Measurements

We measured median total throughput across 10 runs of 100000 requests each using `ab`, with a concurrency of 300 threads. Nginx was configured to serve static web pages filled with random text content of various sizes, using the ECDHE-RSA-AES128-GCM-SHA256 cipher suite with 4096-bit RSA private keys. We used one Nginx worker.

Recall that our kernel changes carry possible performance implications even for non-isolated processes by introducing new logic to the ubiquitous `current` macro. This macro is used very frequently, so even the minimal amount of new code that checks for a possible coprocess causes measurable overhead. Accordingly, we divide our measurements into three groups:

1. Unmodified Nginx/OpenSSL running in an unmodified Linux kernel.
2. Unmodified Nginx/OpenSSL running in our Linux kernel that supports coprocesses.
3. Nginx/OpenSSL with coprocess isolation, running in our Linux kernel that supports coprocesses.

This partitioning enables us to separate the performance impact of the component swapping from that of the increased complexity in the `current` macro.

The results of these measurements are summarized in Figure 6.3 and detailed in Table 6.3, which show each throughput measurement normalized and reported as a percentage of native performance. Nginx/OpenSSL using coprocess isolation achieves up to 95% of the throughput of the non-instrumented Nginx running on an unmodified kernel (native), which is on par with previous systems' performance [72, 38].

However, unlike its predecessors, the coprocess system shows *decreased* performance at larger response body sizes. With 128 KB responses, ERIM achieved essentially the same performance as native [72]. For ERIM, as more time was spent on I/O operations, fewer contexts switches were performed per second, leading to a lower relative overhead. For

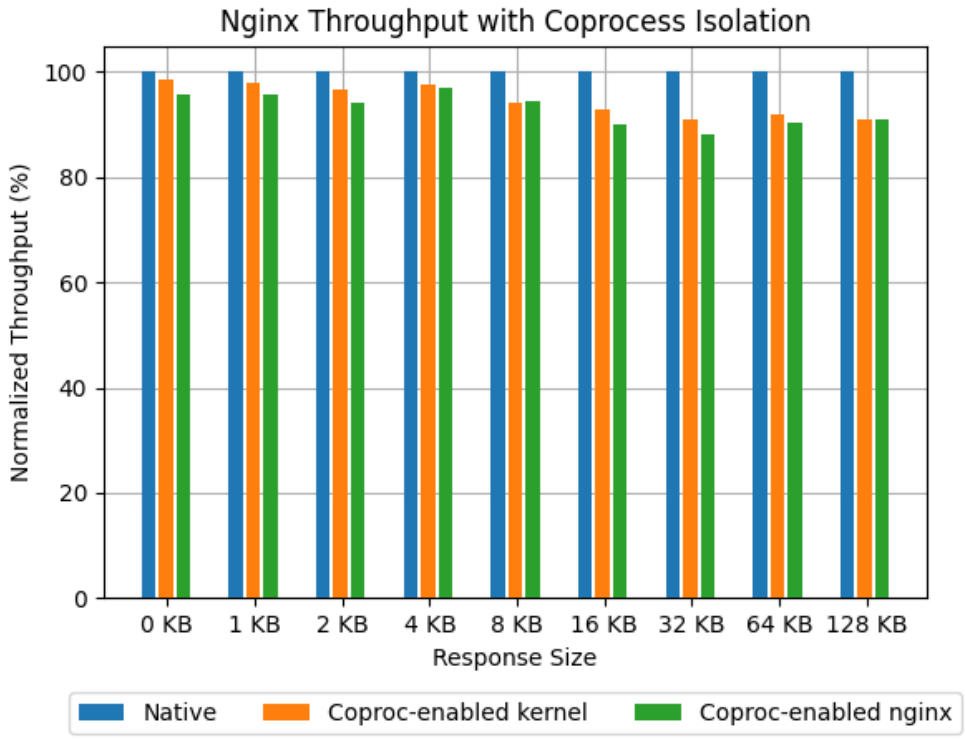

Figure 6.3: Benchmarks comparing Nginx throughput with unmodified kernel; kernel with coprocess support but unmodified Nginx/OpenSSL; and kernel with coprocess support and instrumented versions of Nginx/OpenSSL.

Table 6.3: Throughput of Nginx running with and without coproc instrumentation on a coproc-supporting kernel build, as a percentage of native throughput. Native throughput is measured using an equivalent Nginx build on an unmodified Linux kernel.

Request Size	Native (req/s)	Coproc disabled	Coproc enabled
0 KB	24169	98.7%	95.8%
1 KB	22779	97.9%	95.6%
2 KB	21206	96.7%	94.2%
4 KB	20376	97.6%	97.0%
8 KB	19178	94.2%	94.4%
16 KB	16356	93.0%	89.9%
32 KB	12214	90.9%	88.1%
64 KB	7830	91.9%	90.3%
128 KB	4583	91.1%	91.1%

coprocesses, throughput increases modestly at first as web page sizes increase (up to about 2 KB) then falls and plateaus to about 90% by 64 KB responses. We explain this as a result of the `current` modification overhead, which adds additional overhead to the kernel code that handles the I/O. With more time spent in kernel space reading network data, the added overhead of repeatedly looking up possible coprocesses in `current` overshadows the time saved by fewer context switches. This explanation is consistent with our observation that the gap between the isolated Nginx instance and the non-isolated Nginx instance running in the coproc-enabled kernel does shrink to essentially zero as response size increases. Thus, the entire overhead experienced at the 128 KB response size is entirely due to the kernel changes, not the actual cost of context switching.

6.4 Limitations and Future Work

One limitation of the coprocess design is in the potential for unexpected side effects that result from decoupling intra-process resources by default. We encountered this problem in a bug affecting stored FPU contexts, which we detail in Section 6.2.4. As this example showed, such bugs can also have security implications. Still, we believe that the potential for these edge cases is substantially outweighed by the overwhelming number of possible unintended violations of intra-process security boundaries that would otherwise exist within the kernel.

Our measurements also revealed a substantial performance limitation of this prototype. The added complexity of the oft-used `current` macro incurred measurable overhead, especially for I/O heavy workloads. However, we note that this implementation choice is not central to the coprocess design. Rather, as future work it would be possible to avoid this cost by looking up and storing the appropriate `task_struct` only once on all entries to the kernel.

6.4.1 Simplifying Integration

A significant barrier to adoption for *all* exploit mitigation technologies, arguably even more important than performance overhead, is the complexity of actually integrating with existing software. Developer time is valuable, and organizations may be reluctant to spend much time

on adding new security measures. So, there is a real need for researchers to design mitigations that are efficient not only in terms of computing resources, but also in developer resources.

Library-Level Isolation

One strategy for easing adoption could be to align intra-process components with preexisting units such as dynamic libraries. Dynamically-linked libraries already function as self-contained units to some extent, which may make it easier to isolate them from the main application.

Already, the dynamic linker populates a global offset table (GOT) with function pointers to library routines used by the main executable. All calls from the executable to the library use this GOT. A modified linker could instead inject call gates that perform a context switch before invoking any library procedure, and switch back before a return.

Likewise, memory allocations performed by the isolated library could be intercepted by the linker and placed in an isolated heap. In this way, an entire library could be isolated without any changes at all. It would even be possible to isolate a binary shared library without source code.

For libraries that only expose opaque data structures and don't expect the user to directly access any library-allocated memory, this approach could isolate security-sensitive libraries without modification. Where libraries do need the user to have direct access to some allocated memory, the developer could mark that allocation to use a non-isolated heap.

Library-level isolation does have real challenges. For example, a library that exposes a function as general-purpose as `memcpy` or `printf` is not really isolated at all if the user can invoke these functions with arbitrary arguments and have them execute in the trusted component. But, for libraries that are small and handle security-critical data, library-level isolation may make sense. These libraries would still have to account for the isolation in some ways, such as by treating function arguments as untrusted and validating them accordingly. But, these libraries would not have to concern themselves with the implementation details of any particular form of isolation.

Automatic Instrumentation for Critical Fields

Another possibility is to allow developers to designate only certain security sensitive *fields*, without having to manually locate the code that legitimately accesses them. Instead, a compiler pass could identify pointer dereferences that may access protected fields, and instrument the enclosing function to perform a context switch upon entry and before returning. The developer would then be relieved of the burden of identifying the code of the trusted component manually.

This approach may result in false positives where unnecessary context switches are incurred, but it could still reduce the exposure of security-sensitive fields significantly. Future work could test the feasibility of such a system and measure the security and performance implications of these false positives.

Chapter 7

Conclusion

In-process memory isolation extends traditional system security boundaries to restrict memory accesses between discrete components *within* a single process. Many possible mechanisms have been researched for providing efficient intra-process isolation, but little work has been done on how these systems might be attacked. Recent works propose using a new hardware feature, Protection Keys for Userspace, to implement in-process memory isolation with low context-switching overhead and no execution overhead [72] [38]. We use these systems to explore possible vulnerabilities and challenges in implementing in-process isolation securely.

We find common weakness in both prototypes that we examine, and develop proof-of-concept exploits. Many of the weaknesses stemmed from problems with the W^X assumption, some resulted from kernel confused-deputy attacks, and some were ordinary bugs like race conditions that can be found in any software. Both systems relied on static code invariants—properties of executable code that could be validated at load time and maintained throughout execution. When the W^X assumption is undermined, the static code invariants can be violated. Kernel confused-deputy attacks are challenging to address due to the immense scale of the kernel. The Linux kernel spans millions of lines of code which were all written without a thought towards intra-process isolation, and many interfaces violate the intra-process security model. While known loopholes are mostly easy to fix, a better method is needed to ensure that none are overlooked.

We go on to examine the W^X assumption in the context of another form of intra-process isolation, control-flow integrity (CFI). We show that one of the attacks, the `proc/mem` attack, is also applicable to these systems because they share the same flawed conception of the W^X assumption. Again, we develop a proof-of-concept exploit that shows that this technique works under realistic assumptions in real-world software to bypass *any* fine-grained static CFI with shadow stacks. This technique uses a data-only attack, leveraging file-offset data-oriented gadgets to inject arbitrary executable code into the process at runtime. Such file-offset data-oriented gadgets are present in both Nginx and the GNU standard C library, suggesting that programs using the standard C library for file I/O may be vulnerable to this technique.

In response to the apparent discrepancy between the W^X assumption in theory and in practice, we design and prototype the `xlock` system to instantiate the assumption on Linux. We found that we could eliminate much of the known attack surface by converting all code mappings to be non-writable, private, and anonymous. The remaining interfaces that violate the W^X assumption were discovered and patched through static analysis. We test the `xlock` system on real software and find that it adds no runtime overhead and has a modest memory footprint.

We then synthesize the preceding results to design a hybrid model for intra-process isolation that delegates context switches purely to userspace, but enforces inter-component boundaries in the kernel by transparently treating each component as a different process. When used in conjuction with the `xlock` system, we believe this system mitigates all known vulnerabilities in existing PKU-based intra-process isolation systems. Furthermore, it does so in a systemic way that addresses the root causes of the weaknesses rather than engage in a game of whack-a-mole with individual bugs.

Finally, we compare performance of the coprocess prototype in protecting a web server's private keys. Total throughput of Nginx using coprocess isolation was measured at 90% as compared to native. Unlike ERIM, coprocesses experienced decreased throughput at higher response page sizes. The lower throughput at larger response sizes is accounted for by the added complexity of the `current` macro in the coprocess-enabled kernel. We expect that this inefficiency could be almost completely resolved with additional optimization. We conclude

that coprocesses with `xlock` are a feasible solution for a more robustly secure intra-process isolation system.

Bibliography

[1] CVE-2014-0160. https://nvd.nist.gov/vuln/detail/CVE-2014-0160, 2014. 1

[2] Linux Programmer's Manual: execstack. https://man7.org/linux/man-pages/man8/execstack.8.html, 2017. 5

[3] Linux Programmer's Manual: PROCESS_VM_READV(2). http://man7.org/linux/man-pages/man2/process_vm_readv.2.html, 2017. 24

[4] Kernel.org: Secure Computing with Filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2019. 17, 38, 41

[5] Linux Programmer's Manual: clone(2). http://man7.org/linux/man-pages/man2/clone.2.html, 2019. 80

[6] Linux Programmer's Manual: ptrace(2). http://man7.org/linux/man-pages/man2/ptrace.2.html, 2019. 41

[7] AB - Apache Bench. https://httpd.apache.org/docs/2.4/programs/ab.html, 2021. 84

[8] Incident report on memory leak caused by Cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/, 2021. 1

[9] Nginx Documentation - Module ngx_http_proxy_module. http://nginx.org/en/docs/http/ngx_http_proxy_module.html, 2021. 54, 56

[10] Nginx Project. http://nginx.org/, 2021. 7, 47, 53, 84

[11] OpenSSL Cryptography and SSL/TLS Toolkit. https://www.openssl.org/, 2021. 1, 84

[12] SELinux Project. https://github.com/SELinuxProject, 2021. 5, 70

[13] The LLVM Compiler Infrastructure Project. http://llvm.org/, 2021. 66

[14] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):4, 2009. 5, 6, 12

[15] ARM. TrustZone whitepaper: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2009. 12

[16] Scott Bauer. SROP Mitigation: Signal cookies. Linux Mailing List: https://lwn.net/Articles/674861/, 2016. 32

[17] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. USENIX Association, 2008. 11

[18] E. Bosman and H. Bos. Framing signals - a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, May 2014. 30, 32

[19] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, August 2018. USENIX Association. 21

[20] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019. 6, 7

[21] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic

evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium USENIX Security 19)*, pages 249–266, 2019. 12

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., August 2015. USENIX Association. 6, 12, 21, 49, 58

[23] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 45–58. ACM, 2009. 12, 74

[24] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 5, 2005.

[25] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016. 11, 74

[26] L. Cheng, H. Liljestrand, M. S. Ahmed, T. Nyman, T. Jaeger, N. Asokan, and D. Yao. Exploitation techniques and defenses for data-oriented attacks. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 114–128, 2019.

[27] Jonathan Corbet. Kernel Development. Linux Development Article: http://lwn.net/2001/1011/kernel.php3, 2001.

[28] Thurston HY Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015. 6

[29] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex

Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery. 1, 74

[30] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913. ACM, 2015. 12

[31] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 40. IEEE Press, 2016. 12

[32] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-Process Memory Isolation EXtension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018. 5, 13, 40, 74

[33] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014. 12, 21

[34] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security*, page 2. ACM, 2017. 13

[35] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, volume 17, page 26, 2017. 12

[36] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379. ACM, 2016. 12

[37] Dave Hansen. PATCH 01/33: mm: introduce get_user_pages_remote(). Linux Kernel Patch: https://lkml.org/lkml/2016/2/12/612, 2016. 4, 39

[38] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *2019 USENIX Annual Technical Conference (USENIX ACT 19)*, 2019. 2, 10, 11, 14, 18, 74, 75, 86, 91

[39] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986, 2016.

[40] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 177–192, 2015. 49, 51, 56, 57

[41] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013. 12

[42] Intel. Intel® software guard extensions programming reference: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2014. 12

[43] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2019. 2, 7, 12, 13, 14

[44] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 957–972, 2014. 72

[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, May 2019. 21

[46] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017. 1, 11, 12, 13, 14, 74

[47] Istvan Kurucsai. CVE-2019-5786 Exploit. https://github.com/exodusintel/CVE-2019-5786, 2019. 49

[48] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018. 6

[49] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. 66

[50] Siarhei Liakh, Michael Grace, and Xuxian Jiang. Analyzing and improving linux kernel memory protection: a model checking approach. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 271–280, 2010. 72

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018. 21

[52] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16)*, pages 49–64, 2016. 1, 11, 74

[53] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1607–1619. ACM, 2015. 1, 11, 74

[54] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. ASLR-Guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015. 6, 12

[55] Andy Lutomirski. Rethinking sigcontext's xfeatures slightly for PKRU's benefit? Linux Kernel Mailing List: https://lkml.org/lkml/2015/12/18/571, 2015. 40

[56] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[57] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[58] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997. 12

[59] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. MicroStache: A Lightweight Execution Context for In-Process Safe Region Isolation. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 359–379. Springer, 2018. 1, 13, 74

[60] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque control-flow integrity. In *NDSS*, volume 26, pages 27–30, 2015. 5, 6, 12

[61] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015. 21

[62] James Moris, Stephen Smalley, and Greg Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, pages 17–31. ACM Berkeley, CA, 2002. 17

[63] Micah Morton, Jan Werner, Panagiotis Kintis, Kevin Snow, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Security risks in asynchronous web servers: When performance optimizations amplify the impact of data-oriented attacks. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 167–182. IEEE, 2018. 49

[64] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS 18, page 111–112. Association for Computing Machinery, 2018. 12

[65] Gabriele Paoloni. How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures. *Intel Corporation*, 123, 2010. 84

[66] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM. 6, 74

[67] Saravanan Sinnadurai, Qin Zhao, and Weng fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008. 6

[68] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing SELinux as a Linux security module. *NAI Labs Report*, 1(43):139, 2001. 5

[69] PaX Team. PaX team homepage: https://pax.grsecurity.net/, 2001. 12

[70] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association. 6, 12, 21

[71] Anjo Vahldiek-Oberwagner. ERIM (source code): https://gitlab.mpi-sws.org/vahldiek/erim, 2019. 22, 75

[72] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019. vii, 2, 4, 10, 13, 14, 15, 16, 22, 42, 45, 74, 75, 82, 84, 86, 91

[73] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016. 12

[74] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *ACM SIGOPS Operating Systems Review*, volume 27, pages 203–216. ACM, 1994. 12, 74

[75] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, May 2010. 6, 12, 21

[76] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W. Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1805–1821, Santa Clara, CA, August 2019. USENIX Association. 21

[77] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009. 1, 12, 74

[78] 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 *2013 IEEE Symposium on Security and Privacy*, pages 559–573, May 2013. 6, 21

[79] Jun Zhang, Rui Hou, Wei Song, Zhiyuan Zhan, Boyan Zhao, Mingyu Chen, and Dan Meng. Stateful Forward-Edge CFI Enforcement with Intel MPX. In *Conference on Advanced Computer Architecture*, pages 79–94. Springer, 2018. 12

[80] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 337–352, Washington, D.C., 2013. USENIX. 6, 12, 21

[81] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, 2019. 62, 66

[82] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. On the performance of Intel SGX. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187. IEEE, 2016. 13

Vita

Richard Joseph Connor is from Nashville, Tennessee. He graduated from high school in 2013, and began studies in computer science at the University of Tennessee, Knoxville the following August. He graduated from UT with a B.S. in May of 2016 before starting work as a software security analyst at Cisco Systems. After one year, he returned to UT to pursue an M.S. and PhD in computer science. In May of 2018, he was awarded an M.S. after completing his thesis in applied functional cryptography.