PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems

R. Joseph Connor
University of Tennessee, Knoxville

Tyler McDaniel
University of Tennessee, Knoxville

Jared M. Smith
University of Tennessee, Knoxville

Max Schuchard
University of Tennessee, Knoxville

Abstract

Intra-process memory isolation can improve security by enforcing least-privilege at a finer granularity than traditional operating system controls without the context-switch overhead associated with *inter*-process communication. A single process can be divided into separate components such that memory belonging to one component can only be accessed by the code of that component. Because the process has traditionally been a fundamental security boundary, assigning different levels of trust to components within a process is a fundamental change in secure systems design. However, so far there has been little research on the challenges of securely implementing intra-process isolation *on top of* existing operating system abstractions. We identify that despite providing strong intra-process memory isolation, existing, general purpose approaches neglect the ways in which the OS makes memory and other intra-process resources accessible through system objects. Using two recently-proposed memory isolation systems, we show that such designs are vulnerable to generic attacks that bypass memory isolation These attacks use the kernel as a confused deputy, taking advantage of existing intended kernel functionality that is agnostic of intra-process isolation. We argue that the root cause stems from a fundamentally different security model between kernel abstractions and user-level, intra-process memory isolation. Finally, we discuss potential mitigations and show that the performance cost of extending a `ptrace`-based sandbox to prevent the new attacks is high, highlighting the need for more efficient system call interception.

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.

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. While there are numerous proposals for isolating components within an application, most of them suffer from one of two problems: high overhead during execution, or high cost of switching between isolated components [49]. Recently, two concurrent works, called Hodor [26] and ERIM [49], proposed systems for intra-process memory isolation that achieve dramatically lower overhead by exploiting "memory protection keys for userspace", a new hardware feature available in some recent x86 processors [29]. 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 [49].

PKU 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 [29]. The PKRU register can be written from userspace with the unprivileged `wrpkru` instruction, which 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.

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.

Evaluating the Security of PKU-based Sandboxes. In this paper, 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 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 [25]. 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.

As a result, a secure solution to intra-process isolation requires more careful consideration of all possible attack surfaces in the kernel. We discuss possible mitigations and measure their potential impact on performance. In particular, we show that ERIM's [49] `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 4 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 [49]. 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.

2 Background

2.1 Memory Isolation

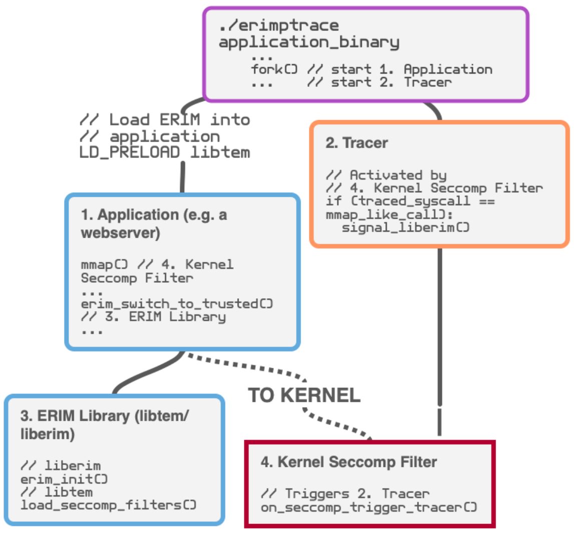
Modern operating systems must provide a stable platform for a complex userspace application ecosystem. For this reason, the kernel carefully restricts interactions between processes. Process-specific resources - including register state and virtual address space - are opaque and inaccessible from other 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 cannot leverage their position to peer into (or modify) another application's private memory space.

Isolation within an application could provide similar benefits. Sensitive data in an application component - e.g. 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 [49] Section 6.5, and [26] 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. Many frameworks have been built for this purpose, with varying costs and characteristics; see Section 7 for a full discussion of these techniques. This paper is primarily concerned with solutions based on Intel Protection Keys for Userspace (**PKU**) [29].

2.2 Intel PKU

The PKU feature (available in Skylake or later Intel server processors since 2017 [29]) regulates memory accesses based on the state of a new 32-bit register, the **PKRU** register [29]. 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 [26, 31, 49]. 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. Tracer
1. Application (e.g. a webserver)
TO KERNEL
3. ERIM Library (libtem/ liberim)
4. Kernel Seccomp Filter

```
./erimptrace
application_binary
    ...
    fork() // start 1. Application
    ...    // start 2. Tracer
```

```
// Load ERIM into
// application
LD_PRELOAD libtem
```

```
// Activated by
// 4. Kernel Seccomp Filter
if (traced_syscall ==
mmap_like_call):
  signal_liberim()
```

```
mmap() // 4. Kernel
Seccomp Filter
...
erim_switch_to_trusted()
// 3. ERIM Library
...
```

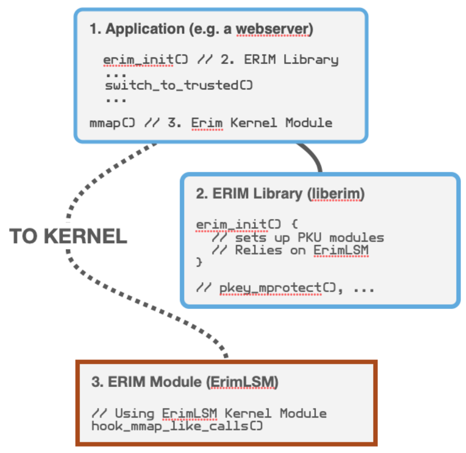
```
// liberim
erim_init()
// libtem
load_seccomp_filters()
```

```
// Triggers 2. Tracer
on_seccomp_trigger_tracer()
```

Figure 1: ERIM Architecture [49] with seccomp filters and process tracing.

2.3 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 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.



```
  erim_init() // 2. ERIM Library
  ...
  switch_to_trusted()
  ...
mmap() // 3. Erim Kernel Module
```

```
erim_init() {
  // sets up PKU modules
  // Relies on ErimLSM
}

// pkey_mprotect(), ...
```

```
// Using ErimLSM Kernel Module
hook_mmap_like_calls()
```

Figure 2: ERIM architecture [49] using a kernel module.

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 [43]), or by installing seccomp filters [2] that inform a tracer process. The seccomp filter with tracer mode is shown in Figure 1, while the kernel mode of operation is depicted in Figure 2. Figures 1 and 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

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`

Table 1: Summary of developed proof-of-concepts.

cause a fault handled by ERIM that signals a scan *for that page alone*. Scan failure results in termination of the program.

2.4 Hodor

As in ERIM, Hodor [26] 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 7. 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.

3 Methodology

3.1 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).

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 [41, 47, 52, 57], they also carry a significant performance penalty and may not be completely effective [13, 21, 53, 55]. 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 [11, 30, 32].

3.2 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.

3.3 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 [48]. 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 [49]. 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 [49].

4 Attacks

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 1. The following sections develop a rough taxonomy of the attacks and present their technical details.

4.1 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," [1] 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 5. 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 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

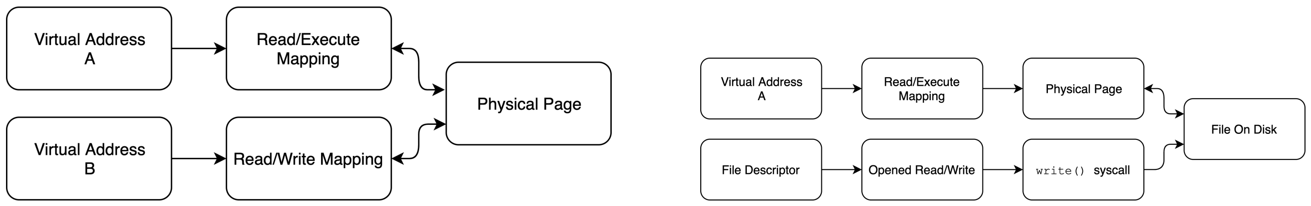


Figure 3: Two examples where W^X does not guarantee code immutability in Linux. Permissions applied to virtual memory mappings do not necessarily apply to the underlying physical memory or file that backs the mapping.

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 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.

4.2 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

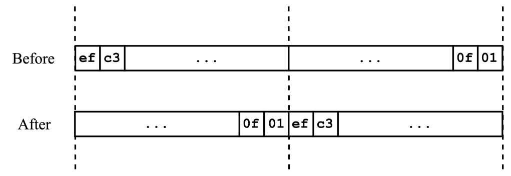


Figure 4: The `mremap` attack. Two pages of code are initially mapped into the process. At this point, the bytes for the `wrpkru` instruction out of order, so the mappings are allowed by the sandbox. A call to `mremap` modifies the page layout to introduce a new `wrpkru` gadget, but does not trigger a new scan.

the sandbox does not monitor or restrict `mremap` calls, the attacker successfully creates a new `wrpkru` gadget without interception by the sandbox. Figure 4 visualizes this attack.

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

4.3 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.

Modifying PKRU via `sigreturn`

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 [10] 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 5 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.

Note also that proposed patches [8] to the Linux kernel that mitigate the `sigreturn`-oriented programming attacks described by Bosman and Bos [10] 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.

4.4 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.

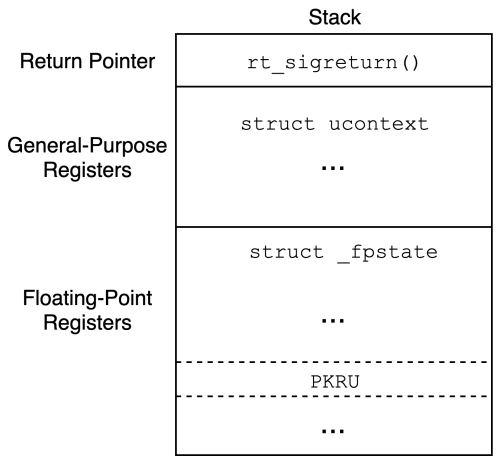


Figure 5: State of userspace stack during signal delivery. The saved PKRU state is stored among the CPU extended state in unprotected memory, and can be modified by the handler or another thread before it is restored by the kernel.

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 4.3 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 4.1) 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.

```
// 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 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 6 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.

4.5 Interfering with Non-memory Shared Resources

Besides attacking memory directly, an attacker may also target other process-wide shared resources that may affect the

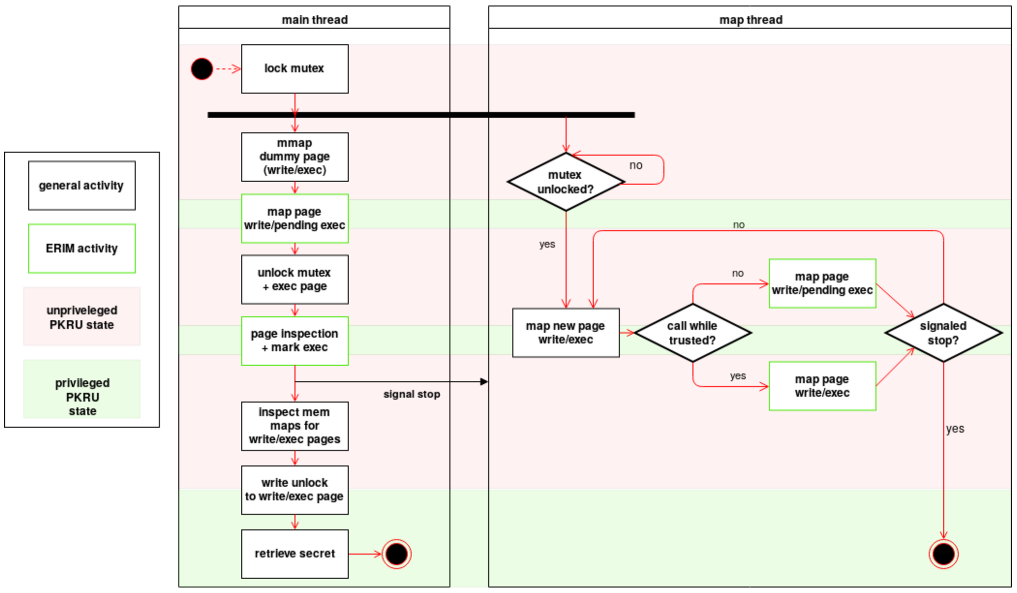


Figure 6: In ERIM, a race condition allows an untrusted thread to make writable and executable mappings, as long as they are made concurrently to a trusted component mapping in another thread.

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 but certain return results take precedent over others (in general, the more restrictive result takes precedence) [2].

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 [2]. 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 4.4 demonstrates this attack in C-like pseudocode.

4.6 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.

5 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 4 by tracing added system calls in ERIM [49]. These system calls are shown in Table 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 4 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 [49] 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 7.

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 7. 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 2 and measured the performance. We find that modifying ERIM's `ptrace`-based sandbox to trace the syscalls responsible for the vulnerabilities in Section 4 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 additional traces applied to mitigate the attacks in Section 4. ERIM-based web servers including the additional traces identified in Table 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. [49].

Figure 8 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 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 4 and shown in Table 2.

6 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 4.1) require a comprehensive solution to guarantee the integrity of executable code in a process's virtual address space, which we discuss later in Section 6.3. 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 6.3. Other attacks, such as the `mremap` and `seccomp` exploits described in Sections 4.2, 4.5, and 4.6 can be miti-

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

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

Table 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.

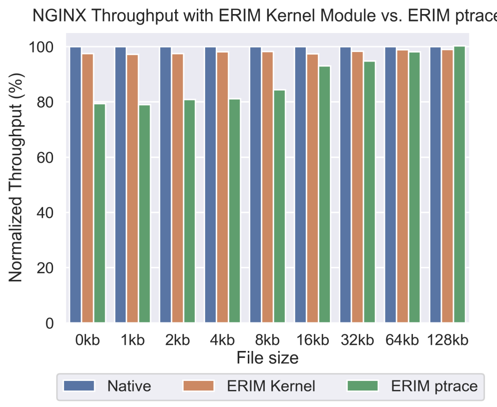


Figure 7: NGINX Throughput (requests/second) with one worker, normalized to native (no protection), ERIM kernel mode vs. ERIM with ptrace.

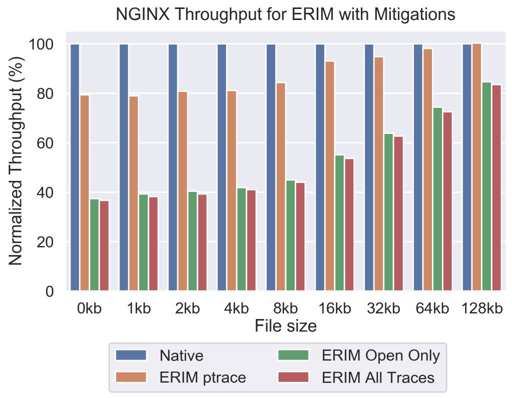


Figure 8: NGINX Throughput (requests/second) with one worker, normalized to native (no protection) of the original ERIM, ERIM with the open() call traced, and ERIM with all syscalls from Table 2 traced, with varying request sizes. Std. deviations all under 2.0%.

gated 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 4.3, 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 [25], 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.

6.1 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 [36]. 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.

6.2 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 [20]. However, as our work demonstrates, the Linux virtual memory system requires non-trivial changes to achieve robust code integrity.

6.3 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 4.1). 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). A purely userspace solution like the `ptrace`-based sandbox would have a much more difficult task of mitigating these issues.

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 [3]. 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.

A mitigation for mutable-backed mappings might be more complicated. Currently, even the shared libraries loaded by the linker are not protected from modification as the process runs (except perhaps by file permissions). Linux once supported a `mmap` flag `MAP_DENYWRITE` which instructed the kernel to prevent any changes to a mapped file, but support for this flag was dropped over concerns about a denial-of-service issue if a user were to map a system file in this way [17]. Despite suggestions that the denial-of-service problem could be mitigated by requiring that the user have permission to execute the mapped file to use this flag, this feature was never implemented (evidently due to a lack of demand). A patch that re-adds support for this flag could help mitigate the file-backed loophole to code integrity.

Closing the file-backed loophole from userspace alone (as in the case of the `ptrace`-based sandbox) seems much more difficult without removing significant functionality or tracing a large number of file I/O-related syscalls. The sandbox might be able to disallow any executable mappings on files not owned by root in the first place, although this would break compatibility with any programs that actually need to load libraries dynamically. Alternatively, a userspace sandbox could monitor I/O calls from the traced process to prevent modification to a mapped executable, but this would carry an unacceptable performance penalty for many applications.

In the case of shared memory, a kernel-based sandbox would need to track permissions on different mappings to the same shared memory. Memory that is mapped writable anywhere must not be allowed to be mapped executable elsewhere. A userspace sandbox may have no safe option but to disallow executable mappings on shared memory in general.

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 5 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 [2]. 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.

7 Related Work

In-process isolation is a well-researched topic with a large body of related work. In this section we summarize other works on in-process isolation and describe where they might be vulnerable to issues similar to those presented in this paper.

Kernel Abstractions. Some proposed systems propose novel kernel abstractions to facilitate low-cost switching between memory views, including shreds [16] and light-weight contexts (lwCs) [33]. 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. Because lwCs can be swapped in userspace, lwC switches are twice as fast as traditional context switches [33] but still 100x more expensive than PKU changes [49]. Because lwCs are described and implemented in FreeBSD rather than Linux, our analysis of the Linux kernel's interaction with intra-process isolation does not apply. Such an analysis for BSD-like systems would be interesting future work.

Wedge [9] 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.

VMFUNC. Intel's VT-x virtualization extensions allow for unprivileged switching between extended page tables. MemSentry-VMFUNC [31], Hodor-VMFUNC [26], and SeCage [34] 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. The trusted/untrusted transition cost of virtualization-based approaches are inexpensive relative to traditional context switches, but more costly than PKU-based equivalents [26]. Attacks that target non-memory process resources (such as in Section 4.3) may apply to these systems.

Static and Dynamic Bounds Checking. Type-safe programming languages [39] 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 unsuited to domains that require direct resource management by the programmer.

Software fault isolation (SFI) [15, 51, 54] can retrofit unsafe languages with similar checks, but at a significant performance penalty. Systems such as NativeClient [54] 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% [31]. Hardware-supported checks (e.g. Intel MPX [29]) offer improved performance but still have significant runtime costs [44], and in some cases are vulnerable to Meltdown-based attacks [12]. SFI techniques also typically require a mechanism for control flow integrity (CFI) [5, 42, 47, 50, 52, 56, 57] 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 [56]. A number of exploits in the literature challenge CFI system security [14, 18, 21].

In general, bounds checking approaches seek to prevent initial exploitation rather than constrain an attacker already executing arbitrary control-flow attacks. For this reason, most of our the attacks in this paper do not directly apply. However, there is one that surprisingly may apply: an attacker may be able to modify memory or code if a vulnerable application can be tricked into writing to `/proc/self/mem`.

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) [35, 46]. While full ASLR can mitigate buffer overflow attacks, an entire family of effective side-channel bypasses [19, 23, 24, 27] casts doubt on the security of such approaches.

Trusted Execution. Trusted enclaves like ARM's TrustZone [7] and Intel's Software Guard Extensions (SGX) [28] 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 [58], and can be vulnerable to side-channel attacks [22].

IMIX [20] and Microstache [40] 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.

PKU. MemSentry-PKU [31] first explored using PKU as an in-process isolation technique, but no provision was made for preventing bypasses by in-process adversaries, so our attacks fall outside their threat model. ERIM [49] and Hodor-PKU [26] hardened the feature to provide isolation in the presence of such adversaries *without* CFI.

ERIM and Hodor's authors considered and effectively blocked several potential exploit avenues. First, trusted component thread stacks must be allocated within trusted memory regions. This separation prevents concurrently executing untrusted threads from spying on trusted thread stacks. Additionally, ERIM's inspection process explicitly considers unauthorized PKRU writes that occur across page boundaries to defend against cross-page `wrpkru` instructions. ERIM installs a non-overridable signal handler for page faults, while Hodor explicitly delays signals during trusted execution.

The Linux kernel's PKU support provides an additional buttress against some exploits. For example, the PKRU is set to the least-permissive (default value) before signals are delivered to processes. While hardware PKRU checks are disabled during privileged execution [29], the kernel checks the current PKRU state before dereferencing userspace pointers [25]. This protection prevents the use of syscalls (e.g. `read`) to circumvent PKU isolation.

8 Conclusion

In-process memory isolation extends traditional system security boundaries to restrict memory accesses between discrete components *within* a single process. 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 [49] [26]. We evaluate the real-world security of proposed PKU-based sandbox designs and make the following contributions:

- We show that both existing PKU-based sandboxes are vulnerable to similar classes of attacks and provide concrete exploits that bypass the isolation to access protected memory from an untrusted component.
- We examine mitigations and performance impacts.
- We analyze the root cause of the design vulnerabilities and suggest that they generally stem from the inconsistency between the threat models and abstractions used by systems researchers and those used by kernel developers.

Future work could further explore the gap between secure system design abstractions and real-world systems by evaluating similar works that create new trust boundaries in an existing system.

Availability
Exploit proof-of-concept software, ERIM software with mitigations, and benchmark software are available at `https://github.com/volsec`.

References

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

[2] Kernel.org: Secure Computing with Filters. `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`, 2019.

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

[4] SELinux Project. `https://github.com/SELinuxProject`, 2019.

[5] 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.

[6] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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, 2015.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

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

[29] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals, 2019.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

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

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

[40] 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.

[41] 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.

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

[43] 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.

[44] 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.

[45] 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.

[46] PaX Team. PaX team homepage: `https://pax.grsecurity.net/`, 2001.

[47] 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.

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.