You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing

Alexios Voulimeneas
imec-DistriNet, KU Leuven
Belgium
alex.voulimeneas@kuleuven.be

Jonas Vinck
imec-DistriNet, KU Leuven
Belgium
jonas.vinck@kuleuven.be

Ruben Mechelinck
imec-DistriNet, KU Leuven
Belgium
ruben.mechelinck@kuleuven.be

Stijn Volckaert
imec-DistriNet, KU Leuven
Belgium
stijn.volckaert@kuleuven.be

Abstract

Memory Protection Keys for Userspace (PKU) is a recent hardware feature that allows programs to assign virtual memory pages to protection domains, and to change domain access permissions using inexpensive, unprivileged instructions. Several in-process memory isolation approaches leverage this feature to prevent untrusted code from accessing sensitive program state and data. Typically, PKU-based isolation schemes need to be used in conjunction with mitigations such as CFI because untrusted code, when compromised, can otherwise bypass the PKU access permissions using unprivileged instructions or operating system APIs.

Recently, researchers proposed fully self-contained PKU-based memory isolation schemes that do not rely on other mitigations. These systems use exploit-proof call gates to transfer control between trusted and untrusted code, as well as a sandbox that prevents tampering with the PKU infrastructure from untrusted code.

In this paper, we show that these solutions are not complete. We first develop two proof-of-concept attacks against a state-of-the-art PKU-based memory isolation scheme. We then present Cerberus, a PKU-based sandboxing framework that can overcome limitations of existing sandboxes. We apply Cerberus to several memory isolation schemes, and show that it is practical, efficient, and secure.

CCS Concepts: • **Security and privacy → Software and application security**; **Systems security**.

Keywords: Security, In-Process Isolation, PKU, Sanboxing

ACM Reference Format:
Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You Shall Not (by)Pass! Practical, Secure, and Fast PKU-based Sandboxing. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3492321.3519560

1 Introduction

Many computer programs contain untrusted components that must be isolated from trusted components to guarantee the confidentiality and integrity of sensitive program state or data. Modern operating systems provide the necessary isolation only at the process boundary. This forces software developers to run components as separate processes (sometimes referred to as compartments) that each have their own virtual address space. One of the drawbacks of process-level compartmentalization is that synchronous interaction between isolated components incurs high performance overhead due to the expensive context-switching required for inter-process communication. The research community has, therefore, proposed several alternative forms of compartmentalization that have better performance characteristics and are often more practical to apply to existing programs. The core idea behind many of these techniques is to isolate untrusted components in-process, thereby allowing them to share typical per-process resources with the rest of the program. Some of these techniques introduced new OS abstractions to set up thread-local address spaces using standard memory management units [12, 20, 38, 51], whereas others leveraged hardware virtualization support [36, 45, 53, 61, 81], ARM memory domains [69], Memory Protection Keys for Userspace (PKU) [19, 33, 34, 36, 45, 48, 63, 69, 70], Supervisor Mode Access Prevention (SMAP) [81], or custom hardware [65] to enable more granular compartmentalization.

Memory Protection Keys for Userspace (PKU) is a hardware feature that is available on recent Intel and AMD server

EuroSys '22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519560

and desktop CPUs [22]. PKU allows programs to be compartmentalized by assigning memory pages to memory protection domains whose access permissions can be set individually by modifying the content of the PKU control register (PKRU). PKU exposes a set of unprivileged instructions to read and modify said register. This allows a program to quickly disable access to all compartments that must be isolated from the currently executing compartment, without having to pay the high cost of the system calls and TLB invalidations required to change page permissions through conventional means. However, an attacker can exploit vulnerabilities, hijack the control-flow of a program, and abuse PKRU-updating instructions to modify PKRU, enabling access to any compartment's memory.

Recently, researchers proposed ERIM [70] and Hodor [36], two efficient PKU-based memory isolation schemes that isolate an application's trusted from its untrusted components by placing them in different memory protection domains. Both systems have built-in sandboxes that prevent adversaries from bypassing the isolation scheme by executing unsafe instructions that modify the PKRU register. Similar to previous work, we refer to these sandboxes as *PKU-based sandboxes* [21]. Unfortunately, these sandbox implementations have known security and usability problems. ERIM, for example, relies on static binary instrumentation (SBI) to neutralize any unsafe instructions in the protected program. However, as SBI cannot reliably distinguish code from data, ERIM could leave some unsafe instructions untouched [82]. Currently, ERIM's sandbox also marks pages that contain unsafe instructions as non-executable, which could lead to usability issues [70]. Hodor's sandbox uses hardware breakpoints to ensure the program cannot execute unsafe instructions. This approach does not rely on SBI like ERIM's, but both systems can still be bypassed using the kernel as a confused deputy [21].

In addition to the aforementioned problems, we discovered two more security flaws in the design of Hodor's sandbox and exploit these flaws in new proof-of-concept attacks that we present in this paper. We then design and implement Cerberus[1], a new PKU-based sandboxing framework that can protect PKU-based memory isolation schemes against all known PKRU-tampering attacks (except for the signal context attacks described in Section 4.2.7). We use Cerberus to develop sandboxes for two existing isolation schemes and evaluate the performance and security of the resulting systems. We conclude that Cerberus enables practical, efficient, and secure PKU-based sandboxing.

In summary, our paper contributes the following:

- We identify new design flaws in Hodor's PKU-based sandbox, and develop two new proof-of-concept attacks that exploit these flaws [36].
- We present Cerberus, a new PKU-based sandboxing framework, and apply our framework to develop sandboxes for two state-of-the-art PKU-based memory isolation schemes: ERIM [70] and XOM-Switch [55]. The resulting sandboxes stop all known attacks (except for the signal context attacks described in Section 4.2.7), including the new attacks we present in this paper [21, 36, 70].
- We perform an extensive evaluation of the constructed sandboxes on real-world server applications and show that Cerberus enables practical, efficient, and secure PKU-based sandboxing.

2 Background

PKU utilizes a new user-mode register (PKRU) to control access rights[2] to memory pages that are tagged with one of 16 available protection keys. The PKRU register is 32 bits wide and has two bits (access disable and write disable) for each key. These bits are checked during memory accesses for all the pages that are associated with a key. The OS provides new system calls, `pkey_alloc` and `pkey_free`, to allocate and free protection keys respectively. A process can tag a page with a key by using the new `pkey_mprotect` system call, and access the PKRU register with unprivileged instructions; `rdpkru` for read and `wrpkru` for write accesses. The `xrstor` instruction can also update the PKRU register if bit 9 in the `eax` register is set prior to the instruction execution.

2.1 PKU-based Memory Isolation Schemes

Some memory isolation schemes leverage PKU to isolate trusted from untrusted components [33, 39, 40, 48, 63]. These systems typically tag memory pages containing trusted code and data with a different protection key than pages containing untrusted code, thereby placing them in different memory protection domains. Trusted code is then allowed to access both domains, whereas untrusted code can only access the untrusted domain. Furthermore, researchers have used PKU to harden JavaScript engines [59], reinforce other exploit mitigations [17, 19, 34, 45], and provide software abstractions for isolation and sandboxing [58].

PKU can also be used to implement eXecute-Only Memory (XOM). XOM is an effective mitigation against advanced code-reuse attacks that rely on reading code [11, 68]. Recent Linux kernels support XOM through an enhanced version of the `mprotect` system call. If a user-space program uses this system call to mark pages as `PROT_EXEC` only, the kernel will assign said pages to a memory domain that has read access disabled and it will update the PKRU register accordingly. XOM support is not yet available in mainstream C libraries and compilers, but can be enabled using XOM-Switch [55]. XOM-Switch's patched dynamic linker and `libc` mark all

[1] Cerberus is available at https://github.com/ku-leuven-msec/The-Cerberus-Project.

[2] The PKRU register only controls data accesses, instruction fetches are not similarly restricted.

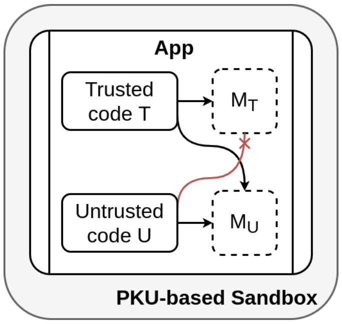

Figure 1. The goal of a PKU-based sandbox is to prevent an attacker that has seized control of U's execution from accessing M_T, while permitting legitimate memory accesses from U to M_U and from T to both M_T and M_U.

pages containing executable code as execute-only using the aforementioned system call.

XOM can easily be bypassed unless it is used alongside mitigations such as CFI. Without CFI, attackers can hijack control flow and disable XOM. XOM-Switch's authors acknowledged this limitation, and proposed to use Intel's Control-flow Enforcement Technology (CET) [2], a recent feature of Intel CPUs, as an effective CFI mechanism. CET enforces hardware-assisted CFI policies using a shadow stack and indirect branch tracking. We show, however, that we can use a PKU-based sandbox instead of CFI to harden XOM-Switch in Section 6.1.2.

2.2 ERIM and Hodor

One major challenge for PKU-based systems is to prevent attackers from tampering with the PKRU register by exploiting a memory vulnerability in the untrusted code and by subsequently locating and executing a PKRU-modifying instruction. One way to prevent such attacks is to apply exploit mitigations to the untrusted code [6, 16, 47]. However, these mitigations can introduce non-trivial run-time overheads, and often cannot fully prevent PKRU tampering [1, 16].

Hodor and ERIM are PKU-based isolation systems that do not rely on such mitigations [36, 70]. Both approaches use two memory domains, M_T and M_U, that contain trusted and untrusted components respectively. Transferring control from trusted components to untrusted components or vice versa happens via well-defined, exploit-proof instruction sequences also known as *call gates*. In addition, Hodor and ERIM each use a sandbox to prevent attackers that manage to compromise an untrusted component from accessing M_T by abusing PKRU-modifying instructions (`wrpkru`, `xrstor`) or system calls like `pkey_mprotect` as depicted in Figure 1.

2.2.1 A Closer Look at ERIM. ERIM compartmentalizes applications into a trusted and untrusted domain. It assumes the trusted components T are not exploitable, but does not make any assumptions about the untrusted components U. ERIM uses call gates to switch between the two domains. Call gates use so-called *safe instructions* (see Listing 1). Safe `wrpkru` instructions are those that are immediately followed by either instructions to validate PKRU's state at run time (lines 12 to 17), ensuring that M_T is locked by PKRU, or by a jump to T (lines 4 and 5). Safe `xrstor` instructions, on the other hand, are immediately followed by instructions that check if bit 9 of eax is set. If one of these run-time validations fails, the control-flow jumps to an instruction sequence that terminates the application. Otherwise, the program execution continues. We refer to any other `wrpkru` and `xrstor` instructions as *unsafe instructions*, as was done in previous work [21]. ERIM's call gates are not exploitable, since they do not contain unsafe instructions. However, an attacker that controls U could abuse unsafe instructions found outside call gates to change PKRU, thereby allowing U to access M_T. Attackers can easily find unsafe instructions because (a) they could appear as operands of other instructions, and (b) x86 instructions do not have a fixed size and the CPU, therefore, allows programs to execute instruction operands as if they were regular instructions themselves.

At startup time, ERIM's PKU-based sandbox scans all executable pages of the protected application using the `/proc/self/mem` interface to verify the absence of exploitable unsafe instructions in M_U pages. Any executable page that contains unsafe instructions is marked as non-executable. Consequently, any attempt to execute code from such a page will trigger a fault that is handled by ERIM's sandbox, and the sandbox terminates the program prematurely. To prevent this, ERIM first uses a static binary rewriter to replace instruction sequences that contain unsafe `wrpkru`s and `xrstor`s by functionally equivalent sequences that do not contain such unsafe occurrences. At run time, the sandbox intercepts and monitors `mmap`, `mprotect`, and `pkey_mprotect` system calls from U that can introduce unsafe instructions or allow access to M_T. ERIM provides two sandbox implementations: one version that is based on `ptrace` [2], and a more efficient one that requires minor kernel modifications.

```
1  xor ecx, ecx
2  xor edx, edx
3  mov TRUSTED_PERM, eax
4  wrpkru // copies eax to PKRU
5  // Jump to trusted code
6
7  // Execute trusted code
8
9  xor ecx, ecx
10 xor edx, edx
11 mov UNTRUSTED_PERM, eax
12 wrpkru // copies eax to PKRU
13 cmp UNTRUSTED_PERM, eax
14 je continue
15
16 syscall exit // terminate program
17 continue:
18 // control returns to untrusted code
```

Listing 1. ERIM's call gate.

2.2.2 A Closer Look at Hodor.

Hodor uses a trusted application loader to partition applications into trusted and untrusted libraries. Hodor's loader ensures that untrusted libraries U can only interact with trusted libraries T through call gates similar to ERIM's. Hodor's PKU-based sandbox monitors the application at run time to stop an attacker from abusing unsafe `wrpkru` instructions that change `PKRU`. When an application attempts to mark a page as executable, the trusted loader first scans the page for unsafe `wrpkru` instructions, and marks the page as non-executable if it contains unsafe instructions. Attempts to execute code from such a page trigger a fault. Upon receiving that fault, Hodor's modified OS kernel attempts to put hardware breakpoints on all unsafe instructions on the same page, and it marks the page as executable. If the page contains more unsafe `wrpkru` instructions than the maximum number of hardware breakpoints the CPU supports, Hodor will single-step through the page instead [2]. This mechanism ensures that all unsafe `wrpkru` instructions will be vetted by Hodor's kernel. When a hardware breakpoint is triggered, Hodor's modified kernel terminates the program execution. Hodor can reclaim hardware breakpoint slots in the debug registers if and when necessary. However, when doing so, it will always mark the pages these breakpoints point to as non-executable.

3 Threat Model

For this paper, we make the following assumptions about the host system, the targeted application, and the attacker. Our assumptions are in line with work in the area [21, 36, 70]:

- **Host System.** We assume Protection Keys for Userspace (PKU) [22] to be available on the target platform and we trust its implementation. The kernel is also considered part of the Trusted Computing Base (TCB).
- **Targeted Application.** We do not make any assumptions about the untrusted code U, but similar to previous work [21, 36, 70], we assume that the initial state of the targeted application is not compromised, and that the PKU-based sandbox is initialized correctly. The trusted code T of the application, however, is considered free of exploitable bugs. We also assume for simplicity that there are only two levels of trust (T and U), and that the application is not using PKU for any purposes, except for memory isolation.
- **Attacker.** We consider an attacker that controls U of the targeted application with the goal to access M_T. For example, an adversary can use code-reuse and control-flow hijacking attacks to exploit unsafe instructions in executable pages of M_U to tamper with `PKRU`'s state, enabling access to M_T. Mitigations like software diversity [47] and CFI [6, 16] raise the bar for such attacks, but we do not rely on such defenses. Attacks that target the underlying hardware such as transient execution [32, 43, 50] and remote-fault injection [13, 41, 66, 71] are considered out of scope for this paper.

4 Challenges

ERIM and Hodor have built-in sandboxes to prevent attackers that control U from accessing M_T by any means. However, there are several security and usability challenges associated with PKU-based sandboxing [21, 36, 70]. We describe these issues below, introduce two new attacks against Hodor, and also discuss potential solutions.

4.1 Handling of unsafe instructions

Executable pages can contain unsafe instructions either because they were put there intentionally by the programmer, or because they are embedded into other instructions (e.g., as an instruction operand). Attackers can exploit these unsafe instructions to tamper with `PKRU`'s state. ERIM and Hodor eliminate or detect unsafe instructions to block such attacks. We show, however, that the proposed techniques to handle unsafe instructions are incomplete.

First, Hodor's sandbox ensures that unsafe `wrpkru` instructions are vetted by its modified kernel, leveraging hardware breakpoints and single-step execution. We carefully examined the open-source implementation of Hodor[3] and discovered that it does not monitor unsafe `xrstor` instructions. Therefore, an attacker that controls U can abuse these unsafe `xrstor` instructions to unlock M_T.

ERIM, on the other hand, relies on SBI to neutralize unsafe `wrpkru` and `xrstor` instructions. In addition, ERIM's sandbox inspects the program at run time to ensure no new unsafe instructions are introduced in the executable pages of M_U. If U tries to map a page that contains unsafe instructions, it is marked as non-executable by ERIM's sandbox to stop attackers from exploiting them. We used the open-source implementation of ERIM[4], repeated the experiments described in the original paper [70], and verified that ERIM's approach works for the tested system (Debian 8, Linux 4.9.60) and applications.

However, we could not replicate the experiments on a recent system (Ubuntu 18.04, Linux 5.3.18), where ERIM failed to eliminate unsafe instructions in binaries such as `ld.so`, `libc.so`, and `libm.so`, either due to SBI's inability to distinguish code from data [82] or due to gaps in ERIM's rewriting rules. Reames showed that constructing a complete set of rewriting rules is a difficult problem [3].

ERIM's sandbox marks pages containing unsafe instructions as non-executable. On recent systems, part of the code of the aforementioned libraries therefore becomes non-executable, which leads even trivial compartmentalized programs to terminate early. We could modify the sandbox to

[3] https://github.com/hedayati/hodor

[4] https://github.com/vahldiek/erim

not mark these pages as non-executable, but this would lead to security issues, since an attacker that controls U can exploit unsafe instructions from these pages.

Proposed Solution: Cerberus (optionally) uses ERIM's binary rewriter to eliminate unsafe instructions, if possible. In addition, Cerberus implements Hodor's vetting scheme in user space to vet the remaining unsafe instructions, and extends it to also deal with unsafe `xrstor` instructions (see Section 5.2).

4.2 PKU Pitfalls

Conor et al. [21] developed proof-of-concept exploits against ERIM and Hodor that bypass their PKU-based sandboxes. These attacks use the kernel as a confused deputy, taking advantage of OS abstractions that are agnostic of PKU-based memory isolation schemes.

4.2.1 Inconsistencies of PT and PKU permissions.

The OS exposes system calls that do not *respect* the enforced page table (PT) and PKU permissions. An attacker can use `process_vm_readv`, `process_vm_writev`, and `ptrace` system calls to directly access M_T from U. PKU-based sandboxes should intercept and monitor these system calls to prevent such accesses. This introduces negligible performance overhead, since these calls are rarely used, as shown in previous work [21].

Another method to circumvent enforced page table and PKU permissions is to use the `procfs` interface. A process can open the `/proc/self/mem` file, which mirrors the process' virtual address space, and perform I/O operations on it. By design these operations ignore page table and PKU permissions, allowing an attacker to directly access M_T from U, or modify non-writable code to tamper with PKRU's state. PKU-based sandboxes should at least intercept and monitor open-like system calls to prevent such attacks. However, it is shown that this adds huge overhead, unless an efficient system call interception and monitoring mechanism is used [21].

Proposed Solution: Cerberus intercepts and monitors `process_vm_readv`, `process_vm_writev`, and `ptrace` system calls similar to previous work [21]. Cerberus also uses a minimal in-kernel syscall agent that efficiently intercepts open-like calls, and denies opening of `/proc/self/mem` file (see Section 5.2).

4.2.2 Mappings with mutable backings.

More problems arise when mapped memory is backed by a mutable file. An attacker can perform direct I/O operations on this file to modify it, regardless of the page table and PKU permissions of the mappings that are backed by it. These modifications are reflected to the corresponding mappings. The OS also allows multiple mappings of the same shared memory with different page permissions that refer to the same physical memory region. Therefore, an attacker can modify an immutable and executable mapping through another writable mapping of the same shared memory. In both mentioned cases, the attacker can add unsafe instructions to executable pages without being detected by the PKU-based sandbox.

Proposed Solution: Cerberus ensures that there are no file-backed mappings, and also imposes restrictions on mapped regions (see Section 5.2).

4.2.3 Changing code by relocation.

Attackers can also introduce unsafe instructions without modifying executable pages of M_U. First, they can allocate two *non-adjacent* memory pages that each contain part of an unsafe instruction at the page boundary. The PKU-based sandbox scans these pages for unsafe instructions and considers both pages safe since they do not contain *complete* unsafe instructions. Attackers can then use the `mremap` system call to move and join the two pages, thereby *forming* an unsafe instruction at the page boundary between the two pages. To stop this attack, the sandbox should re-scan the page boundaries for unsafe instructions after every relocation [21].

Proposed Solution: Cerberus intercepts and monitors `mremap` system call similar to previous work [21].

4.2.4 Influencing intra-process behavior with seccomp.

ERIM and Hodor use the new `pkey_mprotect` system call to isolate M_T from U. The trusted code T allocates a dedicated memory region (M_T) to store secrets such as encryption keys, and uses `pkey_mprotect` to associate it with a different protection domain than U. However, a malicious `seccomp` filter can deny these calls and return a success value, tricking T into storing sensitive data in memory that is not properly isolated from U. A sandbox can prevent this attack by intercepting and restricting `prctl` and `seccomp` system calls [21].

Proposed Solution: Cerberus intercepts and monitors `prctl` and `seccomp` system calls similar to previous work [21].

4.2.5 Modifying trusted mappings.

Attackers can also change the virtual address space to access isolated memory or modify T. For example, an adversary can invoke a `pkey_mprotect` system call to modify the protection key that a page of M_T is tagged with. Hodor prevents such attacks by passing the addresses of T and M_T to the kernel, and denying any attempt to change them from U [36].

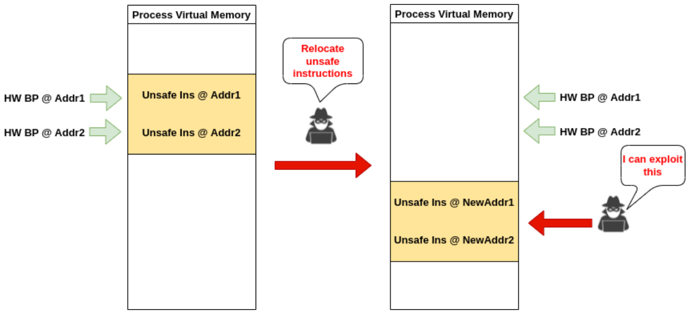

Figure 2. An attacker can abuse the mremap system call to relocate unsafe instructions, *nullifying* instruction vetting based on previously assigned hardware breakpoints.

Proposed Solution: Cerberus forbids pkey_mprotect system call from U (see Section 5.2).

4.2.6 Race conditions in scanning.

PKU-based sandboxes scan executable pages for unsafe instructions. An attacker that controls multiple threads can exploit TOCTTOU race conditions in the memory scanning process to bypass these sandboxes and add unsafe instructions to executable memory. To protect against such attacks, the sandbox should initially mark the page as non-writable and non-executable, scan it, and then mark it as executable only if it does not contain unsafe instructions [21].

Similarly, ERIM assigns newly allocated pages to the same domain they were allocated from. However, because of race conditions in ERIM's ptrace-based sandbox, attackers that seize control of U can trick the sandbox into marking untrusted pages as trusted.

Proposed Solution: Cerberus spawns a unique monitor thread for every application thread, and properly synchronizes the operations that are necessary to determine whether the program is currently executing code in U or T (see Sections 5.2, 5.3 and 6.1).

Proposed Solution: Cerberus does not currently protect against signal context attacks. We discuss ways of extending Cerberus to block such attacks in Section 8.

4.2.7 Signal context attacks.

Conor et al. [21] showed that an adversary can abuse signals to modify PKRU's state without using wrpkru or xrstor instructions. The CPU state, including the PKRU register, is exposed both during the return from a signal handler and during signal delivery. For example, an attacker can craft a CPU state on the stack, and use sigreturn system call to restore an arbitrary value to the PKRU. Conor et al. [21] also showed that it is hard to secure signal handling in multi-threaded applications, since attackers can use the sigaltstack system call to define the signal stack context and exploit race conditions during signal delivery.

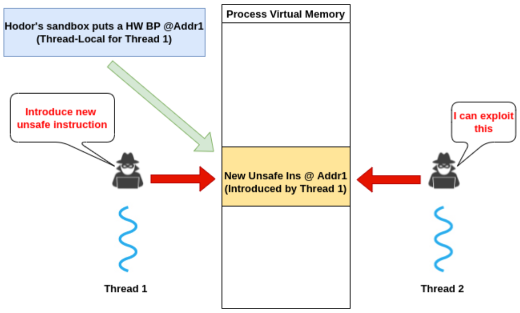
Hodor's sandbox puts a HW BP @Addr1 (Thread-Local for Thread 1)
Process Virtual Memory
Introduce new unsafe instruction
New Unsafe Ins @ Addr1 (Introduced by Thread 1)
I can exploit this
Thread 1
Thread 2

Figure 3. An attacker can use Thread 1 to introduce *new* unsafe instructions, and abuse them from Thread 2, since hardware breakpoints are thread-local.

4.3 New PKU Pitfalls

We examined Hodor's open-source implementation and developed two new proof-of-concept attacks that bypass Hodor's memory isolation scheme and are not stopped by its sandbox. Our attacks specifically target Hodor's instruction vetting mechanism that uses debug registers to monitor unsafe instructions. However, they can be generalized to target other PKU-based sandboxes that use hardware breakpoints to vet unsafe instructions.

4.3.1 Vetted unsafe instruction relocation.

Hodor's modified kernel uses the CPU's debug register to put hardware breakpoints on unsafe instructions and terminates the program if it triggers such a breakpoint. However, Hodor's sandbox does not intercept the mremap system call and, therefore, fails to update breakpoint addresses when the program relocates executable pages as depicted in Figure 2. ***This is a different attack than the one described in Section 4.2.3, and requires additional measures.***

Proposed Solution: Cerberus intercepts mremap system call, and ensures that debug registers are properly updated when a process relocates code.

4.3.2 Incomplete debug register update.

The debug registers Hodor uses for hardware breakpoints are part of the thread context. Hardware breakpoints are, therefore, inherently thread-local. Unfortunately, Hodor only sets breakpoints in the local thread context when it discovers a new unsafe instructions, and does not propagate updated hardware breakpoints to other threads. As a result, an attacker can use one thread to introduce unsafe instructions, and abuse them from another thread to tamper with PKRU's state and unlock M_T to U as depicted in Figure 3.

Proposed Solution: Cerberus augments hardware breakpoints with instruction emulation to vet unsafe instructions in multi-threaded programs (see Section 5.2).

5 Design

With the aforementioned challenges in mind, we designed and implemented Cerberus, a general framework that developers can use to build PKU-based sandboxes. Our goals for Cerberus' design were:

***Goal* 1:** Cerberus should provide the basic components that are necessary to build PKU-based sandboxes.
***Goal* 2:** Cerberus should offer abstractions that allow developers to extend these components.
***Goal* 3:** Cerberus should implement user-space instruction vetting techniques similar to Hodor's. These techniques will allow Cerberus to defend against attacks that bypass the sandbox.
***Goal* 4:** Cerberus should eliminate as many unsafe instructions as possible from the protected program ahead of time. Doing so will ease pressure on the processor's debug registers and improve the performance of the instruction vetting mechanism.
***Goal* 5:** Cerberus should, at all times, be able to determine whether the program is currently executing code in U or T, and be able to determine which memory pages are trusted or untrusted. This is necessary to prevent TOCTTOU attacks, as described in Section 4.2.6.
***Goal* 6:** Cerberus should monitor system calls that can be used to bypass the sandbox (see Sections 4.2.1, 4.2.3, 4.2.4, 4.2.5, 4.2.7, and 4.3.1), but avoid monitoring frequently executed system calls because doing so could substantially reduce the protected application's performance.

5.1 Cerberus' Components

Cerberus contains the following components:

1. **Cerberus Monitor.** A `ptrace`-based monitor that intercepts and monitors system calls. This component is the *core* of the to-be-produced PKU-based sandbox.
2. **Syscall Agent.** A minimal in-kernel syscall agent that forwards *only* the system calls from a configurable list (`SList`) to the Cerberus monitor, while letting the rest to execute without monitoring. The syscall agent also restricts opening of files from a configurable file list (`IList`).
3. **Cerberus Loader.** A custom application loader that initializes the syscall agent, and injects a *special* code page containing a single `rdpkru` instruction into the protected application.
4. **Cerberus APIs.** A rich set of generic macros and functions that developers can use to extend the Cerberus monitor and loader to implement PKU-based sandboxes.
5. **Binary Rewriter.** An SBI tool to eliminate as many unsafe instructions as possible from the protected application ahead of time. Currently, we use ERIM's SBI tool to rewrite binaries [70].

Cerberus decouples the development of PKU-based sandboxes from the development of the underlying PKU-based memory isolation scheme. The first (optional) step of the development workflow is to eliminate a portion of the unsafe instructions that are present in the to-be-sandboxed application with the binary rewriter. We rewrite the application binary, system libraries and other dependencies. Then, the developer uses the Cerberus APIs to extend the default implementations of the Cerberus monitor and loader to build PKU-based sandboxes, tailored to the needs of a particular PKU-based memory isolation scheme. Each produced PKU-based sandbox consists of a modified loader, a modified monitor, and the syscall agent. Figure 4 shows the full development workflow.

Note that Cerberus does not modify the syscall agent. The same syscall agent implementation is used across different sandboxes created with Cerberus. Our framework only provides interfaces to change how the loader and the monitor interact with the syscall agent (see Sections 5.2, 5.3 and 6.1).

5.2 Basic PKU-based Sandbox

Cerberus provides default implementations of its monitor and loader. Along with the syscall agent, these two components form a *basic* PKU-based sandbox. We describe the components and their interactions below, and show how a sandbox developer can use the Cerberus APIs to extend these components and produce PKU-based sandboxes that overcome the challenges described in Section 4.

Initialization. The Cerberus monitor injects the Cerberus loader into the application before launching it. Upon startup, the Cerberus loader maps a *special* executable and non-writable page containing a `rdpkru` instruction into the application's address space. We use this instruction to implement an interface for a `ptrace`-based monitor to read the `PKRU` register (see Section 5.3). The Cerberus monitor ensures that the application cannot unmap the page or change its access permissions. Then, the Cerberus loader initializes the syscall agent through a `prctl` option that we added to the kernel, providing a list of system calls that must be monitored (`SList`) and a list containing the `inode` numbers of sensitive files the application is not allowed to access (`IList`). Both lists are easily configurable through the Cerberus APIs (see Section 5.3).

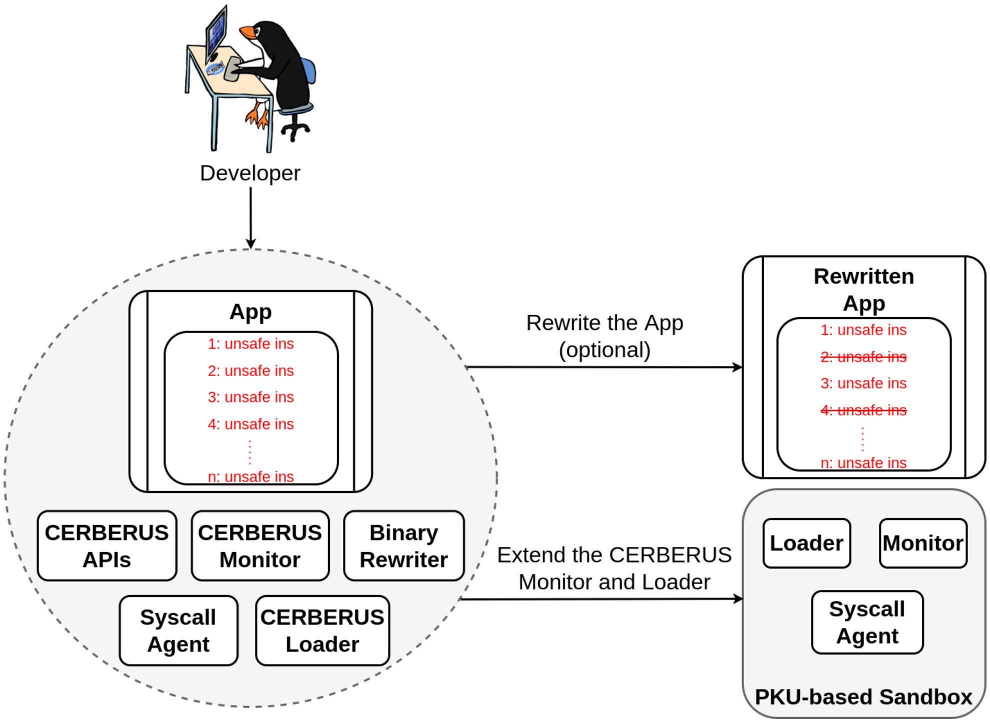

Figure 4. The developer (optionally) uses the binary rewriter to eliminate as many unsafe instructions as possible from the applications. We represent the eliminated unsafe instructions with strike-through text in the rewritten application. Next, the developer uses the Cerberus APIs to extend the Cerberus loader and monitor. The produced PKU-based sandbox consists of the modified loader, the modified monitor and the syscall agent.

System call interception and monitoring. At run time, the syscall agent forwards the system calls included in `SList` to the Cerberus monitor. System calls that are not in the list can be executed without monitoring. The agent also intercepts open-like calls directly in kernel space and denies the application's requests to open sensitive files. By referring to sensitive files using their `inode` number, our agent can also block access to hard or symbolic links that refer to sensitive files.

The Cerberus monitor also ensures that the syscall agent is only initialized *once* after `execve` or `fork/clone`-like system calls. Currently, `IList` only contains the `inode` of `/proc/self/mem` to protect against attackers that aim to abuse these files to bypass memory isolation (see Section 4.2.1). Blocking opening of sensitive files can also be directly implemented in a `ptrace`-based monitor like the Cerberus monitor, but it would add substantial run-time overhead, as shown in previous work [21].

The Cerberus monitor intercepts and monitors the following system calls: `modify_ldt`, `prctl`, `seccomp`, `ptrace`, `process_vm_readv`, `process_vm_writev`, `mprotect`, `pkey_mprotect`, `pkey_alloc`, `pkey_free`, `mmap`, `munmap`, `mremap`, `execve`, `shmat`, `shmdt`, and all variants of the `fork/clone` system.

Whenever the Cerberus monitor needs to determine if the program is currently executing code in U or T, it temporarily moves the program's instruction pointer to the `rdpkru` instruction on the special page mapped by the loader. The monitor then forces the application to execute this single instruction, reads the PKRU value from the application's register context, and then restores the original register context. We implemented this mechanism because the `ptrace` API currently does not offer any options to read the contents of the PKRU register directly, and because reading said contents is necessary to prevent attacks such as those described in Section 4.2.6.

The Cerberus monitor assumes that the application always executes code in U. The developer should use the Cerberus APIs to define the PKRU values that correspond to U and T respectively, and extend the Cerberus monitor (see Sections 5.3 and 6.1).

An attacker that controls U can use system calls to access M_T directly or to unlock it to U (see Sections 4.2.1, 4.2.4 and 4.2.5). To prevent such attacks, the Cerberus monitor forbids the following system calls from U: `modify_ldt`, `prctl` setting `seccomp`, `seccomp`, `ptrace`, `pkey_alloc`, pkey_free, pkey_mprotect, shmat and shmdt. It also rejects process_vm_readv and process_vm_writev system calls that attempt to access M_T from U.

The Cerberus monitor does not implement a mechanism to reliably track pages which are in M_T. Consequently, M_T is empty by default. The developer should use the Cerberus APIs to build such a mechanism (see Sections 5.3 and 6.1).

Handling of unsafe instructions. The Cerberus monitor scans all executable pages U loads into the address space at run time for unsafe instructions. To do so, it intercepts and monitors mmap, mremap and mprotect system calls from U as these calls could introduce unsafe instructions. We implemented a modified version of Hodor's instruction vetting scheme in user space using ptrace to deal with the discovered unsafe instructions. Unlike Hodor, however, our scheme vets *both* unsafe wrpkru and xrstor instructions. In addition, Hodor's sandbox always terminates execution whenever a hardware breakpoint is triggered, while our monitor allows execution of unsafe xrstor instructions to continue if bit 9 of the eax register is not set. If the bit *is* set, we terminate the program. The monitor also implements techniques described in Sections 4.2.3 and 4.3.1 to deal with attacks that use code relocation to introduce unsafe instructions. Eliminating unsafe instructions using binary rewriting reduces *debug register pressure* and, thus, decreases the likelihood that we have to resort to single-step execution when a page contains more unsafe instructions than the number of debug registers (see Section 2.2.2).

The Cerberus monitor considers all wrpkru and xrstor instructions as unsafe, except those included in a configurable list (AllowList). The AllowList is empty by default. The developer should use the Cerberus APIs to modify the AllowList (see Sections 5.3 and 6.1).

Dealing with *dangerous* mappings. Similar to previous work [21], the Cerberus monitor enforces W^X by intercepting and monitoring system calls that map pages or change page permissions, since Linux, by default, allows pages that are both writable and executable. Moreover, the Cerberus monitor does not allow executable mappings that are MAP_SHARED or MAP_SHARED_VALIDATE to protect against attackers that try to modify an immutable mapping via another mutable mapping of the same shared memory (see Section 4.2.2).

Dealing with attacks that try to directly modify the underlying object of a file-backed mapping is more complicated, though. First, the Cerberus monitor intercepts system calls that map memory (e.g., mmap) and replaces the file-backed mappings, with MAP_ANONYMOUS ones, ensuring there are no mappings that are backed by a file. Then the Cerberus monitor copies the file contents that the application initially attempted to map to the mapped region. To prevent attacks through memory pages with mutable backing files (see Section 4.2.2), the Cerberus monitor imposes restrictions on mapped regions. For example, we reject any attempt to map pages that are simultaneously MAP_SHARED and executable. Although we did not observe any compatibility or usability issues that arose from these restrictions, we discuss their potential implications in Section 8.

Protecting multi-threaded programs. Protecting multi-threaded programs is challenging because hardware breakpoints are only set in the thread-local register context and because malicious threads could attempt to modify executable code while it is being scanned for unsafe instructions. This opens new possibilities for attacks that would not be possible in single-threaded programs, as we explain in Sections 4.2.6 and 4.3.2. To counter these threats, we spawn a unique Cerberus monitor object and thread for every application thread we protect. The Cerberus monitors that supervise threads of the same process share data structures and *always* enter a critical section when they scan the application memory for unsafe instructions. The Cerberus monitors must enter the same critical section when an application thread attempts to execute a system call that could change the contents of any of the pages that are being scanned (e.g., mremap). This design avoids the problem of race conditions during memory scanning.

Secondly, whenever a process spawns a thread for the first time (i.e., when the program transitions from single-threaded to multi-threaded execution), the Cerberus monitors stop relying solely on hardware breakpoints for instruction vetting, since updates to the set of breakpoints would not propagate beyond the currently executing thread as described in Section 4.3.2. Instead, the Cerberus monitors *only* use hardware breakpoints to vet the unsafe instructions whose addresses were stored in the debug registers at the moment the process started to use multiple threads, whereas code pages that contain unsafe instructions that are not protected by debug registers are marked as non-executable. These pages include those that were mapped as non-executable by the Cerberus monitor before switching to multi-threaded execution, as well as new pages mapped by the threads. Attempts to execute instructions from code pages that were marked as non-executable by the Cerberus monitor trigger a fault. When the Cerberus monitor is notified of faults on one of these pages, which it can determine by inspecting the instruction pointer, it does not terminate execution immediately. First, the Cerberus monitor checks if the instruction that was about to get fetched is an unsafe instruction or not. If it is an unsafe wrpkru instruction, the Cerberus monitor terminates execution, while if it is an unsafe xrstor, the Cerberus monitor terminates execution *only* if bit 9 of register eax is set. Otherwise, the instruction is considered safe and the Cerberus monitor uses an emulation engine for x86 instructions (included in Cerberus) to emulate the instruction, essentially updating the program's state to make it seem like the instruction was *actually* executed.

At run time, the Cerberus monitor can decide if emulation of instructions is necessary by detecting if multiple threads are used. To do so, it intercepts system calls that create and destroy threads, and checks the /proc/self/task file. Even though emulating instructions in user space is slow, we did not experience significant performance degradation in our experiments that included multi-threaded applications, since unsafe instructions are rare as described in previous work [36, 70]. In addition, eliminating a portion of unsafe instructions with SBI decreases the number of instructions that should be emulated, since the monitor needs to mark fewer code pages as non-executable. We provide details for the emulation engine in Section 6, and we discuss alternative ways to deal with multi-threaded applications in Section 8.

5.3 The Cerberus APIs

Cerberus provides a rich set of abstractions to facilitate development of PKU-based sandboxes. We describe below the most important abstractions below.

Configuration of the syscall agent. Developers can use simple macros to modify the SList and IList the Cerberus loader passes to the syscall agent. For example, we can add a *new* system call to the SList by adding CERBERUS_MASK_-SET(SYSCALL_NUMBER) in the Cerberus loader.

Definition of U and T. Developers can use simple macros to define the values of PKRU that correspond to U and T respectively. These macros are based on code from previous work [36, 70].

Identification of the current executing domain. Cerberus provides a ptrace-based interface to read PKRU's state, reliably identifying the currently executing domain. We described the functionality of this interface in Section 5.2.

System call handling. Similar to previous work [76, 78], Cerberus provides macros to insert hooks before and after the execution of certain system calls. In addition, our framework exports macros and functions for accessing registers and memory, and a rich logging infrastructure. Developers can use these abstractions to write system call handlers. For example, developers can build a mechanism that maintains a list of memory pages belonging to M_T by writing handlers for the pkey_mprotect and mprotect system calls (see Section 6.1).

Modification of the AllowList. Developers can use macros and functions to add instruction sequences that include wrpkru and xrstor to the AllowList. The Cerberus monitor's memory scanning considers these instruction sequences safe. This allows us to recognize sequences such as ERIM's call gates. This code is based on previous work [36, 70].

Emulation Engine. Cerberus exports macros and functions to emulate x86 instructions. We use instruction emulation to protect multi-threaded programs (see Section 5.2).

6 Implementation and Use Cases

Cerberus consists of an SBI tool, a monitor, a loader, a set of APIs, and the syscall agent (see Figure 4). We used ERIM's binary rewriter as our SBI tool, and we implemented the other Cerberus components on top of ReMon, a security-oriented Multi-Variant Execution Environment (MVEE) [76]. The implementations of Cerberus's monitor, loader, APIs, and syscall agent comprise ≈1 KLOC, ≈650 LOC, ≈9.4 KLOC, and ≈79 LOC of C/C++ code respectively. Along with ≈229 LOC in shell scripts, this gives us a total of ≈11 KLOC.

The Cerberus APIs include an extensible emulation engine that currently supports 170 x86 instructions and that comprises ≈2752 LOC of C/C++. We based its implementation on code written for a related project, making only minor modifications [75]. We implemented our syscall agent as a small kernel patch for Linux kernel 5.3.18.

6.1 Use Cases

We applied the development workflow described in Section 5.1 to build PKU-based sandboxes for ERIM and XOM-Switch.

6.1.1 Use Case 1 – A Sandbox for ERIM.

First, we defined the same PKRU values as ERIM for U and T, and we also added ERIM's call gates to the AllowList. Second, we implemented a mechanism that tracks which pages are in M_T by intercepting pkey_mprotect system calls from T. To do so, we used the Cerberus APIs to modify how the Cerberus monitor handles the pkey_mprotect system call. Specifically, the monitor identifies the current executing domain by checking PKRU's value, and if it corresponds to T, it inspects pkey_mprotect's arguments. ERIM's T *only* uses pkey_mprotect to tag pages of M_T with T's protection key. Consequently, this mechanism reliably tracks pages of M_T. We did not have to make any other changes to the basic PKU-based sandbox described in Section 5.2. Our total implementation effort for this use case was limited to ≈55 LOC of C/C++ code.

Hodor's memory isolation scheme is similar to ERIM's. Consequently, we would only have to make minor changes (e.g., defining slightly different PKRU values for U and T) to our ERIM sandbox to apply it to Hodor is well.

6.1.2 Use Case 2 – A Sandbox for XOM-Switch.

We also used Cerberus to build a sandbox for XOM-Switch. To the best of our knowledge, this is the first PKU-based sandbox for XOM-Switch. This sandbox prevents attackers from abusing PKU to disable XOM, and thus, eliminates the requirement of an additional defense like CFI to

APP	ERIM-CPI (No Sandbox)	ERIM-CPI with the Cerberus Sandbox	ERIM-SS (No Sandbox)	ERIM-SS with the Cerberus Sandbox
nginx (1 worker)	6.54%	4.97%	0.57%	1.91%
nginx (2 workers)	5.72%	6.20%	4.62%	3.32%
nginx (3 workers)	1.55%	2.23%	-1.22%	1.05%
lighttpd (1 worker)	–	–	0.16%	2.14%
lighttpd (2 workers)	–	–	0.22%	0.23%
lighttpd (3 workers)	–	–	-0.01%	-0.03%
geometric mean	3.87%	4.10%	0.71%	1.43%

Table 1. We isolated shadow stacks and safe regions in CPI/CPS with ERIM (ERIM-SS and ERIM-CPI respectively). We measured the overhead of standalone ERIM-SS and ERIM-CPI when they are not protected by a sandbox (no sandbox) compared to the native execution. We report the overhead of ERIM-SS and ERIM-CPI with the Cerberus sandbox compared to the native execution. The Cerberus sandbox here corresponds to the sandbox described in Section 6.1.1. – indicates that the experiment failed. The standard deviation is below 1.75% in all cases.

protect XOM (see Section 2.1). We treated XOM-Switch as a PKU-based memory isolation scheme, in which U includes all the code. This is the default behavior of the basic PKU-based sandbox, so a developer does not need to define the PKRU values that correspond to U and T (see Section 5.2). Similarly, we do not need to modify the AllowList, since XOM-Switch does not rely on user space wrpkru instructions for inter-domain transitions (see Section 2.1). Consequently, the sandbox considers all wrpkru and xrstor unsafe (see Section 5.2). We implemented a mechanism that tracks pages which are in M_T by intercepting mprotect system calls. Within our mprotect handlers, we assign all execute-only pages to M_T and we block any further changes to an execute-only page's permissions, thus blocking the attacks described in Section 4.2.5. This effort took only ≈16 LOC of C/C++ code.

7 Evaluation

We evaluated the performance and the security of the PKU-based sandboxes created with Cerberus.

7.1 Performance

We ran our experiments on an HP Z6 G4 workstation with a 12-core Intel Xeon Silver 4214 CPU running at 2.20 GHz and 64 GB of RAM (Turbo-Boost and Hyper-Threading were disabled). The machine runs Ubuntu 18.04.6 LTS with version 5.3.18 of the Linux kernel. We applied a minimal kernel patch that implements the syscall agent. We evaluate the constructed PKU-based sandboxes on popular high performance server applications: nginx, lighttpd and redis. We ran a benchmarking client on a separate machine that is connected to the workstation through a gigabit Ethernet connection. The client machine has a 6-core Intel Core i7-8700K CPU running at 3.70 GHz and 64 GB of RAM (Turbo-Boost and Hyper-Threading were disabled). The client machine runs Ubuntu 18.04.6 LTS with version 5.4.0 of the Linux kernel. For nginx and lighttpd, we used wrk benchmark to request a 4KB page for 10 seconds over 10 concurrent connections. For redis, we used redis-benchmark, distributed with Redis, with the default workload (100000 requests and 50 parallel connections).

For the experiments described in Section 7.1.2, the client communicates with the server over HTTPS, while for the experiments described in Sections 7.1.1 and 7.1.3 communication happens over HTTP. We measured the throughput of the server applications running under our defense relative to the throughput of the native execution. We ran each experiment 10 times, removed the highest and lowest values as outliers, and reported the average of the 8 remaining values. We configured lighttpd and nginx to use 1–3 workers, and redis to use 1–3 I/O threads. The server applications can saturate the network connection when configured with 3 workers (lighttpd, nginx) and 3 I/O threads (redis). As a result, we did not try configurations with more than 3 workers and I/O threads.

The developed PKU-based sandboxes, described in Section 6.1, identified unsafe instructions in nginx, redis, ld.so, libm.so and libc.so during our experiments. We eliminated a portion of unsafe instructions with ERIM's SBI tool. However, we could not neutralize *all* unsafe instructions with this tool. We did not investigate further the reason that the tool failed, since the constructed sandboxes do not solely rely on the SBI being successful in removing all unsafe instructions.

7.1.1 Protecting safe regions in CPI/CPS and SS.

Similar to previous work [70], we used ERIM to isolate safe regions of CPI/CPS [46]. We changed ≈13 LOC to fix an LLVM bug and to port the CPI compiler of ERIM to our testing environment (Ubuntu 18.04 with kernel 5.3.18). In the same manner, we used ERIM to isolate the safe regions of a shadow stack implementation (SS for short) [17]. To do so, we added ≈38 LOC to the SS compiler passes to add ERIM's functionality. We refer to the above compiler passes as ERIM-CPI and ERIM-SS respectively. We applied ERIM-CPI to nginx and ERIM-SS to lighttpd and nginx. We could

APP	ERIM-OpenSSL with the Cerberus Sandbox
nginx (1 worker)	1.12%
nginx (2 workers)	0.66%
nginx (3 workers)	1.21%
lighttpd (1 worker)	0.44%
lighttpd (2 workers)	0.31%
lighttpd (3 workers)	0.49%
redis (1 I/O thread)	-1.29%
redis (2 I/O threads)	1.34%
redis (3 I/O threads)	0.01%
geometric mean	0.47%

Table 2. We isolated OpenSSL keys in server applications with ERIM (ERIM-OpenSSL). We report the overhead of ERIM-OpenSSL with the Cerberus sandbox compared to the native execution. The Cerberus sandbox here corresponds to the sandbox described in Section 6.1.1. The standard deviation is below 2.67% in all cases.

APP	XOM-Switch with the Cerberus Sandbox
nginx (1 worker)	0.04%
nginx (2 workers)	-0.02%
nginx (3 workers)	-0.09%
lighttpd (1 worker)	0.02%
lighttpd (2 workers)	0.50%
lighttpd (3 workers)	0.16%
redis (1 I/O thread)	1.48%
redis (2 I/O threads)	0.88%
redis (3 I/O threads)	0.00%
geometric mean	0.33%

Table 3. We applied eXecute Only Memory (XOM) using XOM-Switch. We protected XOM-Switch with the Cerberus sandbox. The Cerberus sandbox here corresponds to the sandbox described in Section 6.1.2. We report the overhead of XOM-Switch with this sandbox compared to the native execution. The standard deviation is below 2.17% in all cases.

not run `lighttpd` after applying ERIM-CPI to it because of CPI's imprecise handling of aliasing relations between memory references. We also verified that `lighttpd` fails with the original CPI compiler [46]. Similarly, we could not run `redis` after applying either ERIM-CPI or ERIM-SS. Again, we verified that `redis` also fails after compilation with the original CPI [46] and SS [17] compiler. Consequently, we concluded that it is not our code that is responsible for the failures.

We show the overhead for each successful experiment in Table 1. For the experiments, we removed ERIM's sandbox (no sandbox) or replaced it with the sandbox described in Section 6.1.1 (Cerberus sandbox). For ERIM-CPI with Cerberus sandbox we report overhead of 2.23–6.20% with geometric mean of 4.10%, while for ERIM-SS with the Cerberus sandbox we report overhead of -0.03–3.32% with geometric mean of 1.43%. For standalone ERIM-CPI we report overhead of 1.55–6.54% with geometric mean of 3.87%, while for standalone ERIM-SS with we report overhead of -1.22–4.62% with geometric mean of 0.71%. The results for standalone ERIM-CPI and ERIM-SS are consistent with previous work [81]. We measured the overhead of standalone ERIM-CPI and ERIM-SS without the protection of any sandbox to show that even in the worst case (6.20%), most of the overhead (5.72%) can be attributed to the PKU-based memory isolation scheme and not the sandbox.

7.1.2 Isolating OpenSSL keys in server applications.

Similar to previous work [70], we isolated OpenSSL session keys in popular server applications with ERIM (ERIM-OpenSSL), to protect against server application vulnerabilities such as Heartbleed [27]. We configured `lighttpd`, `nginx` and `redis`, through their config files, to use ERIM-OpenSSL and only use ECDHE-RSA-AES128-GCM-SHA256 cipher and AES encryption for sessions. For the experiments, we replaced ERIM's sandbox with the sandbox described in Section 6.1.1 (Cerberus sandbox). Our results are shown in Table 2. We report an overhead of -1.29–1.34% with geometric mean of 0.47%, which is lower compared to previous work that was evaluated on a similar setup [70].

7.1.3 Protecting Execute Only Memory.

We used XOM-Switch [55] to apply eXecute Only Memory (XOM). XOM-Switch is vulnerable to attackers that attempt to abuse PKU to disable XOM, unless it is combined with an additional mitigation such as CFI. We lift this requirement by combining XOM-Switch with the PKU-based sandbox described in Section 6.1.2 (Cerberus sandbox). Our results are depicted in Table 3. We report overhead of -0.09–1.48% with geometric mean of 0.33%. Our results indicate that it is more efficient to protect XOM with a PKU-based sandbox than CET, since concurrent work showed that CET imposes overhead of 2–8% [15].

7.2 Security and Completeness

We analyzed the security of the constructed sandboxes on existing proof-of-concept attacks [21] and the two additional attacks that we discovered while building Cerberus (see Section 4.3). For the former, we used the open-source implementation of the exploits[5], while for the latter we used our proof-of-concept exploits. We verified that the developed sandboxes stop all the attacks described in Sections 4.2 and 4.3, except for signal context attacks (see Section 4.2.7). This is not a fundamental limitation of our approach, but of the current prototypes of the framework and the constructed sandboxes. We discuss potential solutions to stop signal context attacks in Section 8. Our results are depicted in Table 4.

[5] https://github.com/VolSec/pku-pitfalls

Attack	Cerberus Sandboxes
Inconsistencies of PT permissions [21]	✓
Inconsistencies of PKU permissions [21]	✓
Mapping with mutable backings [21]	✓
Changing code by relocation [21]	✓
Influencing intra-process behavior with seccomp [21]	✓
Modifying trusted mappings [21]	✓
Race conditions in scanning [21]	✓
Determination of trusted mappings [21]	✓
Signal context attacks [21]	✗
Vetted unsafe instruction relocation Section 4.3.1	✓
Incomplete debug register update Section 4.3.2	✓

Table 4. Security analysis of the PKU-based sandboxes described in Section 6.1. We refer to these sandboxes as Cerberus sandboxes. ✓ indicates that the sandbox stops the attack, while ✗ indicates the opposite.

Sandbox Characteristics	Hodor's Sandbox	ERIM's Sandbox	Cerberus Sandboxes
Handling of unsafe instructions	Incomplete	Incomplete	Complete
Kernel Modifications	Major	Minor	Minor
PKU Pitfalls Protection	✗	✗	✓*
New PKU Pitfalls Protection	✗	N/A	✓
Performance overhead	Low	Low	Low

*Except for signal context attacks (see Section 4.2.7).

Table 5. Comparison of different PKU-based sandboxes. We refer to the sandboxes described in Section 6.1 as Cerberus sandboxes.

We compared the sandboxes described in Sections 6.1.1 and 6.1.2 (we refer to them as Cerberus sandboxes) with the ones provided by ERIM and Hodor. The comparison is shown in Table 5. The Cerberus sandboxes are the first PKU-based sandboxes that can handle unsafe instructions without causing security or usability issues (see Section 4.1). The Cerberus sandboxes are also the first PKU-based sandboxes that prevent the attacks described in Sections 4.2 and 4.3, except for signal context attacks (see Section 4.2.7). The new attacks described in Section 4.3 specifically target Hodor's instruction vetting mechanism, and are not applicable (N/A) to ERIM. Both the Cerberus sandboxes and ERIM's sandbox are implemented in user space; nevertheless they require a small kernel patch to optimize performance. Last, all the sandboxes in this list incur low performance overhead.

8 Discussion

In this section, we discuss the limitations of our approach and alternative solutions.

Signal context attacks. A concurrent work with ours proposes Endokernel [15], a subprocess virtualization scheme to deal with challenges in in-process isolation. Specifically for signals, the authors describe a signal virtualization mechanism that prevents attackers from tampering with the PKRU register by abusing signals (see Section 4.2.7). However, two of the three provided implementations of Endokernel rely on additional mitigations, software diversity and CFI, which come with their own limitations regarding efficacy and performance. Some of the techniques used in Endokernel could be applied to our framework to prevent signal context attacks. This is feasible because `ptrace` can intercept, delay, deny, and redirect signals in ways similar to Endokernel [15].

Restrictions on memory mappings. Cerberus imposes restrictions on memory mappings to deal with attackers that target mappings with mutable backings as described in Section 6.1.1. This might lead to usability issues in applications such as older JIT engines, which used double-mapping as a way to bypass SELinux's W^X policy [26, 57]. Modern JIT engines no longer use double-mapping, however, since it can be detrimental to the application's security [23]. In case we have to allow such mappings, we could use static and dynamic techniques to intercept *all* shared memory accesses [75], and block attackers from abusing multiple mappings of the same physical memory to introduce unsafe instructions.

Syscall interposition. Similar to previous work [76, 77], we could extend our syscall agent to forward system calls to an in-process monitor instead of a `ptrace`-based monitor. This would allow the sandbox to efficiently interpose frequently-called system calls in user space (e.g., open-like calls). To deal with attacks that target the in-process monitor, we could implement the in-process monitor in a safe language like Rust, or use PKU to isolate the in-process monitor from the untrusted code [64, 80]. Alternatively, we could also extend Cerberus to support alternative system call interposition techniques that leverage binary rewriting [5, 8, 37], virtualization [10, 44], or syscall user dispatch [4].

Multi-threaded applications. Currently, we use instruction emulation to deal with applications that use multiple threads. However, this might add significant overhead in cases where several pages contain unsafe instructions. One suitable alternative could be to use process-wide events monitoring that will be available in future kernels [28]. Otherwise, we could implement a *stop-the-world* mechanism using signals to ensure that all the threads of a process are stopped, and then update their debug registers synchronously. This would require though to first provide a concrete solution for signal context attacks (see Section 4.2.7).

Multiple levels of trust. We only experimented with application partitioning schemes that use two levels of trust (trusted and untrusted). However, with our framework it would be possible to develop sandboxes for systems that use more than two memory domains, since the constructed sandboxes can reliably determine the current executing domain by inspecting PKRU as described in Section 5, and track the sensitive data of each domain.

9 Related Work

In-process isolation has been explored in depth, resulting in dozens of systems. In this section, we summarize works on in-process isolation that do not rely on PKU.

OS abstractions. Previous work introduced OS abstractions to enable multiple memory views and fast transition between them within a process address space [12, 20, 38, 51]. These approaches expose thread-like entities, control which resources they can access, and permit efficient transitions between them. However, all these techniques are not directly applicable to legacy code without code modifications.

Virtualization-based techniques. Hodor [36] and SeCage [53] leverage virtualization extensions [22] (VT-x), to provide different memory views for trusted and untrusted code. SeCage is vulnerable to an in-process adversary unless it is used in conjunction with CFI, while Hodor's VT-x based implementation is less efficient than Hodor's PKU-based counterpart [36]. Intel and AMD CPUs provide Supervisor-mode Access Prevention (SMAP) hardware feature to disable kernel accesses to user space memory. Seimi uses SMAP and VT-x to provide low cost and secure in-process isolation [81]. xMP [61] extended Xen hypervisor's `altp2m` subsystem [49, 60] and the Linux memory management system to isolate sensitive user space and kernel data in disjoint xMP memory domains.

Hardware extensions. Researchers also proposed hardware extensions to provide efficient fine-grained component isolation. CHERI [83] and CODOMs [74] extended the RISC and x86 ISAs respectively, with capabilities. Donky [65], on the other hand, augmented the x86 and RISC-V ISAs to provide secure memory protection domains similar to PKU. MicroStache [56] and IMIX [31] extended the x86 ISA with instructions to access safe regions. ARM memory domains [7] are similar to PKU domains, but they are only available on 32-bit chips and domain permissions can only be modified with privileged instructions. This paper focuses on solutions that can be built on commodity x86 CPUs.

SFI. Software fault isolation (SFI) restricts parts of an application code from accessing memory outside of designated bounds [18, 24, 25, 29, 30, 54, 67, 79, 85, 86]. SFI techniques employ complex static and dynamic analysis and instrumentation that introduce non-negligible overhead. In addition, many of the proposed techniques rely on an additional mitigation such as CFI, to prevent in-process attackers from bypassing bounds checks.

Compartmentalization. Partitioning an application into compartments and defining which resources they can access is an open problem and it is orthogonal to this paper. Previous work focuses on identifying suitable isolation boundaries in applications and OSes using automatic and semiautomatic (e.g., annotations) techniques [9, 14, 33, 35, 42, 52, 62, 72, 73, 84]. However, completely automating compartmentalization of existing software is still challenging.

10 Conclusion

Recent research has explicitly highlighted the extreme care that should be taken when implementing PKU-based sandboxing, mentioning a large number of edge cases and a difference in perspective between the OS and the security community on PKU as contributing factors. In this paper, we analyzed the various challenges of PKU-based sandboxing. We also introduced two new proof-of-concept attacks that bypass Hodor's sandbox.

We then presented Cerberus, a new PKU-based sandboxing framework that facilitates development of PKU-based sandboxes. We applied our framework to build sandboxes for two state-of-the-art PKU-based memory isolation systems: ERIM and XOM-Switch. We evaluated the security and performance of the constructed sandboxes using proof-of-concept exploits and high-performance server applications respectively. Our extensive evaluation shows that Cerberus overcomes limitations of existing work, enabling practical, efficient, and secure PKU-based sandboxing.

Acknowledgments

We thank our reviewers and especially our shepherd, Gaël Thomas, for their valuable suggestions and comments, that substantially helped in improving the paper. We also thank Adriaan Jacobs, Dokyung Song, Kostis Kaffes, Anjo Vahldiek-Oberwagner, Marios Kogias, and Zhe Wang for helpful discussions, and Per Larsen, Paul Kirth, Karel Dhondt, Jorn Lapon, and André Rösti for feedback on earlier drafts of this paper. This research is partially funded by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity.

References

[1] 2016. Close, but No Cigar: On the Effectiveness of Intel's CET Against Code Reuse Attacks. https://grsecurity.net/effectiveness_of_intel_cet_against_code_reuse_attacks

[2] 2021. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[3] 2021. Unintended Instructions on x86. Philip Reames – A collection of (public) notes on assorted topics. https://github.com/preames/public-notes/blob/master/unintended-instructions.rst

[4] Last accessed 2022. Syscall User Dispatch. The kernel development community. https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html

[5] Last accessed 2022. syscall_intercept. System call intercepting library: https://github.com/pmem/syscall_intercept.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS).*

[7] ARM. 2015. ARM Memory Domains. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0211k/Babjdffh.html.

[8] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. 2022. SaBRe: load-time selective binary rewriting. *International Journal on Software Tools for Technology Transfer* (2022), 1–19.

[9] Markus Bauer and Christian Rossow. 2021. Cali: Compiler-Assisted Library Isolation. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*.

[10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Andrea Bittau, Adam Belay, Ali José Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *IEEE Symposium on Security and Privacy (S&P)*.

[12] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[13] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. CAn't touch this: Software-only mitigation against Rowhammer attacks targeting kernel memory. In *USENIX Security Symposium*.

[14] David Brumley and Dawn Song. 2004. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *USENIX Security Symposium*.

[15] Im Bumjin, Yang Fangfei, Tsai Chia-Che, LeMay Michael, Vahldiek-Oberwagner Anjo, and Dautenhahn Nathan. 2021. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. *arXiv preprint arXiv:2004.04846* (2021).

[16] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 16.

[17] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *IEEE Symposium on Security and Privacy (S&P)*.

[18] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast Byte-Granularity Software Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[19] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. 2022. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *USENIX Security Symposium*.

[20] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *IEEE Symposium on Security and Privacy (S&P)*.

[21] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[22] Jonathan Corbet. 2015. Intel Memory Protection Keys. https://lwn.net/Articles/643797/.

[23] Jan de Mooij. 2015. W^X JIT-code enabled in Firefox. https://jandemooij.nl/blog/wx-jit-code-enabled-in-firefox/.

[24] Liang Deng, Qingkai Zeng, and Yao Liu. 2015. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP International Information Security and Privacy Conference*.

[25] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. Pnacl: Portable native client executables. *Google White Paper* (2010).

[26] Ulrich Drepper. 2006. SELinux Memory Protection. https://akkadia.org/drepper/selinux-mem.html.

[27] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *Internet Measurement Conference (IMC)*.

[28] Marco Elver. 2021. Add support for synchronous signals on perf events. https://lwn.net/Articles/848984/.

[29] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[30] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-Level Sandboxing on the X86. In *USENIX Annual Technical Conference*.

[31] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX: In-Process Memory Isolation EXtension. In *USENIX Security Symposium*.

[32] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. 2017. Lazarus: Practical side-channel resilient kernel-space randomization. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[33] Adrien Ghosn, Marios Kogias, Mathias Payer, James R. Larus, and Edouard Bugnion. 2021. Enclosure: Language-Based Restriction of Untrusted Libraries. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[34] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. 2021. IskiOS: Intra-kernel Isolation and Security using Memory Protection Keys. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[35] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. 2015. Clean Application Compartmentalization with SOAAP. In *ACM Conference on Computer and Communications Security (CCS)*.

[36] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *USENIX Annual Technical Conference*.

[37] Petr Hosek and Cristian Cadar. 2015. Varan the unbelievable: An efficient n-version execution framework. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[38] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *ACM Conference on Computer and Communications Security (CCS)*.

[39] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. 2021. VIP: Safeguard Value Invariant Property for Thwarting Critical Memory Corruption Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[40] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Dawu Gu, Hang Zhang, Siqi Ma, Zhiyun Qian, and Juanru Li. 2022. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *IEEE Symposium on Security and Privacy (S&P)*.

[41] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. 2020. V0ltpwn: Attacking x86 processor integrity from software. In *USENIX Security Symposium*.

[42] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Stijn Volckaert, and Michael Franz. 2022. PKRU-Safe: Automatically Locking Down the Heap Between Safe and Unsafe Languages. In *European Conference on Computer Systems (EuroSys)*.

[43] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*.

[44] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*.

[45] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *European Conference on Computer Systems (EuroSys)*.

[46] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Security Symposium*.

[47] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*.

[48] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. 2022. FlexOS: Towards Flexible OS Isolation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[49] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System. In *Annual Computer Security Applications Conference (ACSAC)*.

[50] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[51] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[52] Shen Liu, Gang Tan, and Trent Jaeger. 2021. Ptrsplit: Supporting general pointers in automatic program partitioning. In *ACM Conference on Computer and Communications Security (CCS)*.

[53] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *ACM Conference on Computer and Communications Security (CCS)*.

[54] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture.. In *USENIX Security Symposium*.

[55] Daiping liu Mingwei Zhang, Ravi Sahita. 2018. eXecutable-Only-Memory-Switch (XOM-Switch). In *Black Hat Asia Briefings (Black Hat Asia)*.

[56] Lucian Mogosanu, Ashay Rane, and Nathan Dautenhahn. 2018. MicroStache: a lightweight execution context for in-process safe region isolation. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[57] John Richard Mose. 2006. Virtual Machines and Memory Protections. https://lwn.net/Articles/210272/.

[58] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys. In *USENIX Annual Technical Conference*.

[59] Taemin Park, Karel Dhondt, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. 2020. NoJITsu: Locking Down JavaScript Engines. In *Symposium on Network and Distributed System Security (NDSS)*.

[60] Sergej Proskurin, Tamas Lengyel, Marius Momeu, Claudia Eckert, and Apostolis Zarras. 2018. Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection. In *Annual Computer Security Applications Conference (ACSAC)*.

[61] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P. Kemerlis, and Michalis Polychronakis. 2020. xMP: Selective Memory Protection for Kernel and User Space. In *IEEE Symposium on Security and Privacy (S&P)*.

[62] Nick Roessler, Lucas Atayde, Imani Palmer, Derrick McKee, Jai Pandey, Vasileios P Kemerlis, Mathias Payer, Adam Bates, Jonathan M Smith, Andre DeHon, et al. 2021. μSCOPE: A Methodology for Analyzing Least-Privilege Compartmentalization in Large Software Artifacts. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[63] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. 2021. CubicleOS: A Library OS with Software Componentisation for Practical Isolation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[64] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium*.

[65] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *USENIX Security Symposium*.

[66] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. In *BlackHat USA*.

[67] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*.

[68] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*.

[69] Zahra Tarkhani and Anil Madhavapeddy. 2020. μTiles: Efficient Intra-Process Privilege Enforcement of Memory Regions. *arXiv preprint arXiv:2108.03705* (2020).

[70] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. In *USENIX Security Symposium*.

[71] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. 2016. Drammer: Deterministic rowhammer attacks on mobile platforms. In *ACM Conference on Computer and Communications Security (CCS)*.

[72] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. 2017. Towards Fine-Grained, Automated Application Compartmentalization. In *Workshop on Programming Languages and Operating Systems*.

[73] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M Smith. 2018. BreakApp: Automated, Flexible Application Compartmentalization.. In *Symposium on Network and Distributed System Security (NDSS)*.

[74] Lluís Vilanova, Muli Ben-Yehuda, Nacho Navarro, Yoav Etsion, and Mateo Valero. 2014. CODOMs: Protecting software with Code-centric memory Domains. In *International Symposium on Computer Architecture (ISCA)*.

[75] Jonas Vinck, Bert Abrath, Bart Coppens, Alexios Voulimeneas, Bjorn De Sutter, and Stijn Volckaert. 2022. Sharing is Caring: Secure and Efficient Shared Memory Support for MVEEs. In *European Conference on Computer Systems (EuroSys)*.

[76] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication.. In *USENIX Annual Technical Conference*.

[77] Alexios Voulimeneas, Dokyung Song, Per Larsen, Michael Franz, and Stijn Volckaert. 2021. dMVX: Secure and Efficient Multi-Variant Execution in a Distributed Setting. In *European Workshop on System Security (EuroSec)*.

[78] Alexios Voulimeneas, Dokyung Song, Fabian Parzefall, Yeoul Na, Per Larsen, Michael Franz, and Stijn Volckaert. 2020. Distributed heterogeneous N-variant execution. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[79] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*.

[80] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *European Workshop on System Security (EuroSec)*.

[81] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *IEEE Symposium on Security and Privacy (S&P)*.

[82] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in X86 Binaries. In *Proceedings of the 2011th European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*.

[83] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *IEEE Symposium on Security and Privacy (S&P)*.

[84] Yongzheng Wu, Jun Sun, Yang Liu, and Jin Song Dong. 2013. Automatically partition software into least privilege components using dynamic data dependency analysis. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[85] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (S&P)*.

[86] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: fully verified software fault isolation. In *ACM international conference on Embedded software*.