Universität des Saarlandes
MI Fakultät für Mathematik und Informatik
Department of Computer Science

Master's Thesis

Execute-Only Memory as a Security Hardening Feature on x86-64

submitted by

Tristan Hornetz

on April 16, 2024

Reviewers

Dr. Michael Schwarz

Prof. Dr. Thorsten Holz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, 16. April 2024,

(Tristan Hornetz)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 16. April 2024,

(Tristan Hornetz)

Abstract

Execute-Only Memory (XOM) is a rarely used, but versatile memory protection scheme, in which instruction fetches are permitted, but data reads and writes are not. In the context of x86_64, it is mainly used in defensive schemes against code-reuse attacks. Besides this however, there is very little research on applications that could benefit from its unique memory protection capabilities. In my master's thesis, I therefore investigate the characteristics of XOM, with the primary goal of identifying novel application scenarios. To this end, I present a set of software libraries that make XOM available to user-mode programs and use them to conduct studies on XOM's performance aspects and potential attack vectors.

The results of this effort are two key observations: Firstly, XOM proves to be highly resistant against transient execution attacks like Spectre and Meltdown. XOM can thus serve as a component of low-cost mitigation schemes against them. Secondly, it is possible to use XOM to hide cryptographic secrets from privileged local attackers. This may provide an alternative to Trusted Execution Environments on platforms where such facilities do not exist, with potential applications in Digital Rights Management.

Acknowledgements

First and foremost, I would like to thank my advisor, Lukas Gerlach, who guided me throughout all parts of the writing process and provided valuable feedback. Without your active support, this work would likely not have been possible. Furthermore, I want to express my gratitude towards my reviewers, Dr. Michael Schwarz and Prof. Dr. Thorsten Holz, for taking the time and agreeing to review this thesis. Finally, I would like to thank my family, especially my parents, for their personal encouragement and tireless support throughout all of my studies. Thank you all!

Contents

Chapter 1

Introduction

Execute-only Memory (abbreviated *XOM*, sometimes also called Execute-no-Read memory) describes memory areas from which instruction fetches are permitted, but data reads and writes are not. Although this concept is quite simple, it is rarely utilized in practice, as hardware support on modern platforms is limited. Today, its most prevalent use is in embedded systems, where hardware manufacturers seek to protect their firmware from reverse-engineering attempts by preventing direct code disclosure [1].

On x86-based architectures, XOM sees virtually no practical use at all. While it is an integral component of many defensive schemes against code-reuse attacks [2–8], these schemes are mostly academic in nature and have yet to see popular adoption. Apart from this, there is very little research on applications for XOM on x86_64. This is surprising, given XOM's relative simplicity and high versatility. Therefore, this thesis seeks to evaluate the properties and security guarantees of XOM on x86_64 on a more fundamental level, with the primary goal of identifying novel use cases and application scenarios. This involves a study of XOM's performance implications and an evaluation of potential attack vectors against XOM. Additionally, I present a set of software libraries that make XOM available to user-mode programs on Linux, utilizing hardware features like *Memory Protection Keys (MPK)* and Intel's *Extended Page Tables (EPT)*.

The results of this effort are promising: As XOM allows for storing secrets without making them readable, it is a surprisingly effective countermeasure against transient execution attacks like Spectre and Meltdown. Virtually all Spectre-like attacks require their victim to have read access permissions to the target information, which is not the case with XOM. Meltdown-like attacks can reveal secrets that are cached in specific CPU-internal buffers, but no buffer that is directly influenced by instruction fetches is known to be susceptible. If adequately implemented, XOM-based defenses can thus mitigate these attacks.

The second key observation of this thesis is that XOM enforced by EPT can provide

exceptionally strong security guarantees, which have yet to be fully utilized in defensive schemes. As EPT is a component of Intel's hardware virtualization extensions, its page table entries are managed by the hypervisor, which means that guests by themselves are unable to read secrets in XOM if the hypervisor does not allow for this. Consequently, attackers cannot directly disclose the protected code, even if they fully compromise the guest kernel. Yet, code in XOM is executable, and cryptographic secrets stored within it, e.g., as immediate values, can thus still be processed. This is a security guarantee typically only provided by Trusted Execution Environments (TEEs), which are not as widely supported on consumer-grade processors as EPT [9].

Therefore, this thesis also explores methods of utilizing EPT-enforced XOM to perform cryptography without disclosing secret keys to the guest, in hopes that these methods can serve as a TEE replacement where such hardware facilities are unavailable. This broad concept is called XOM-based *Key Locking*.

As part of this effort, I propose two novel concepts, which along with XOM, serve as the foundation for Key Locking: *Page Locking* and *Register Clearing*. With Page Locking, the hypervisor forcibly overwrites XOM pages before making them readable to guests again during cleanup. Once a secret is "locked" into XOM, it thereby becomes fully inaccessible to data reads, even after XOM pages become readable again. This allows guest kernels to "forget" cryptographic secrets while preserving the capability to use them by executing the code.

Register Clearing ensures that the kernel cannot retrieve these secrets from the register state while they are in use. This is enforced by the hypervisor, which fully or partially overwrites the registers when handling interrupts in an XOM page, thereby preventing the guest kernel from inspecting the program's internal state. While programs need special recovery mechanisms to handle these register clearing events gracefully, this is not difficult to implement in practice. To show that this approach is practical, I provide implementations of AES-128-CTR and HMAC-SHA256 that can not only recover from register clearing but also employ a range of defensive programming techniques to defend against privileged attackers, thereby making key disclosure virtually impossible.

To help contextualize the significance of these results, this thesis also aims to identify concrete applications that may benefit from XOM. Notable among these are Digital Rights Management (DRM) systems, which may utilize Key Locking as a hardening tool when enforcing the copyright of remotely distributed media files. Other potential use cases include protection against reverse engineering, similar to the concept found in embedded systems, and even providing tamper resistance for online video games.

Finally, this work also discusses the limitations of XOM in x86_64. For example, it demonstrates that EPT-enforced XOM can incur runtime overhead at the microarchitectural level, even if the protected software is not modified. Moreover, one particular attack called Interrupt-driven Code Recovery enables privileged adversaries to reconstruct

protected code with a high degree of accuracy.
In summary, the primary contributions of this thesis are:

- The integration of EPT-XOM allocation into the Xen hypervisor. This implementation is made publicly accessible in hopes that it can aid future research.
- XOM-based Key Locking, a method of hiding encryption keys from privileged attackers without utilizing a TEE.
- An evaluation of the performance implications of XOM on x86_64, demonstrating that EPT-enforced XOM by itself can cause a non-negligible runtime overhead.
- The assessment of potential attacks against XOM's security guarantees, highlighting its resistance to transient execution attacks.
- The identification of previously unexplored applications of XOM in diverse security contexts.

Chapter 2

Background

2.1 Execute-Only Memory

Although it is relatively uncommon in the context of contemporary computer systems, XOM is hardly a new concept. Early implementations even predate the advent of the microprocessor, with it being supported by the Multics operating system for the GE 645 mainframe computer [10]. The intended use case for this feature in Multics is privacy preservation, for example by preventing unauthorized access to the code of a classroom grading program.

While similar privacy-preserving techniques remained the singular use of XOM in the following years, the growing accessibility and complexity of computer systems led to an eventual diversification in XOM's potential applications. For example, Yarvin et al. propose a method to increase the performance of inter-domain procedure calls by mapping the memory of multiple domains into one 64-bit address space [11]. In this scheme, segmentation is achieved through anonymity, meaning domains are unaware of the other domains' locations in virtual memory and can thus not access them. Also, scanning the address space for other domains is impractical due to its large size. To still allow for controlled inter-process communication, a publicly known 'intermediary' code section redirects calls between domains, typically in the form of a jump table. However, to prevent the destination addresses from leaking, this code section requires protection from unauthorized read accesses, which necessitates the use of execute-only memory.

Another application for execute-only memory is the protection of intellectual property. For instance, Lie et al. propose an execute-only scheme based on memory encryption [12], which aims to prevent unauthorized redistribution and manipulation of software. In their suggested XOM architecture, specialized cryptography hardware sits between external memory and the processor's instruction decoder, allowing the processor to decrypt the

instruction stream while loading it from memory. When executing an encrypted program on this architecture, the processor first loads the program's unique key from a special header section. This key is itself encrypted with an asymmetric cipher, the private key of which is only known to the processor's microcode. Hence, only the processor can decrypt the program, whereas a third party can only encrypt new code. While this approach does not prevent a potential attacker from reading or overwriting the memory contents, it makes it exceedingly difficult to redistribute or manipulate the code in any meaningful way, thus achieving the goal of enabling execution while denying direct access. Memory encryption has since seen widespread adoption as a component of trusted execution environments [13] and secure virtualization techniques [14], but its use for creating execute-only memory remains uncommon due to its performance overhead and extensive hardware requirements [15, 16].

More recently, XOM was proposed as a countermeasure against code-reuse attacks [2]. For exploitation techniques like return-oriented programming to be feasible, an attacker must have access to the code of the target binary. In scenarios where this is not the case, they must hence first mount a code-disclosure attack, which execute-only memory can effectively prevent. It has therefore seen use in mitigation schemes for JIT-ROP [17], Blind-ROP [18], and other exploitation techniques involving code-disclosure. XOM also sees use as a component of *leakage-resistant diversity* schemes, which combine XOM with code diversification techniques to mitigate code reuse for binaries that are known to potential attackers (see Section 7.2).

2.2 Protection Keys

Despite its many applications, support for XOM on the x86_64 architecture remains limited. The only way to create execute-only regions on a native, non-virtualized x86_64 system without employing software-implemented fault handlers is through the *Memory Protection Keys (MPK)* feature [9, 19], which exists on some Intel CPUs since the introduction of the Skylake architecture, and on some AMD CPUs since the introduction of Zen 3.

On CPUs supporting MPK, 4- and 5-level page table entries contain a 4-bit field called the *Protection Key*, each possible value of which is associated with a configurable set of restrictions on how a page can be accessed. Software can modify these restrictions in a special 32-bit register called *pkru*, which can be read and written to using the *rdpkru* and *wrpkru* instructions respectively. In this register, each of the 16 possible protection keys has a write-disable bit and a read-disable bit. Applications can configure these bits to control reads and writes to pages tagged with a specific protection key. However, instruction fetches are not affected by this, so setting both bits for a given protection

key results in XOM.

What sets MPK apart from more traditional memory protection mechanisms is that its configuration is accessible to user-mode applications. Access to the *pkru* register is not restricted by privilege level, allowing an application to enforce its own memory segmentation rules without having to invoke the kernel. MPK is therefore also known as the *Protection Keys for User-Space (PKU)* feature.

From a performance-oriented standpoint, this concept makes sense: In a database containing highly sensitive data such as bank accounts, an out-of-bounds write access caused by a programming error could have disastrous consequences. It is therefore useful to restrict access to memory regions containing such data, permitting modifications only for a short time when explicitly necessary. However, enforcing these restrictions using traditional mechanisms would require at least two kernel invocations per transaction, causing an unacceptable performance overhead. MPK solves this issue by reducing the effort needed to modify the permissions to a single register access. Hence, MPK is most commonly found on CPUs aimed at data centers, which are more likely to run applications benefiting from such a mechanism.

Unfortunately, since MPK as a concept is designed to solve safety issues rather than security issues, employing it for any security-related purpose is challenging. Initial attempts to achieve secure intra-process memory isolation with MPK-based sandboxes [20, 21] were quickly shown to be susceptible to a variety of powerful attacks, some of which are still unaddressed by more recent techniques [22]. These attacks are possible, among other factors, due to inconsistent enforcement of MPK permissions in the kernel, mutable file-backed memory regions, and potential abuse of the `sigreturn` mechanism [23]. XOM schemes based on MPK are susceptible to the same weaknesses, and should therefore be treated with caution. However, due to the low performance cost associated with using MPK, and its relative simplicity, it can still be highly useful in the context of a weak attacker model, or as a redundancy layer in a multi-level security concept. As discussed later in this thesis, it can also serve as a defense mechanism against transient execution attacks (see Section 6.1.2).

2.3 Hardware-assisted Virtualization

To fully virtualize a system, a hypervisor must provide a virtualization environment in which even kernels that are unaware of being virtualized can successfully execute. However, this comes at a steep performance cost with traditional virtualization techniques, in which the guest kernel is executed with user-mode privileges.

On many older systems, the most popular virtualization technique is therefore an alternative method called *para-virtualization (PV)*. This scheme uses a hypervisor-specific API

(called para-API) to run expensive to virtualize tasks in the hypervisor's context rather than the guest's [24], thus drastically increasing virtualization performance. However, this scheme also has a detrimental impact on compatibility, as a given hypervisor's para-API is typically only supported by a small selection of guest kernels.

To resolve this trade-off between compatibility and performance, CPU manufacturers today are integrating facilities into their processors that can handle most virtualization tasks in hardware, thereby eliminating the need for a para-API altogether [9, 19]. Although implemented nearly identically by both Intel's and AMD's CPUs, these extensions have different names depending on the manufacturer, being called *virtual-machine extensions (VMX)* by Intel and *secure virtual-machine (SVM)* by AMD. For consistency, mechanisms added by VMX and SVM under different names are referred to using Intel's VMX naming convention for the remainder of this thesis.

At its core, VMX introduces a mode of operation for VM guests named *VMX non-root operation*, which from the perspective of a guest kernel, is nearly indistinguishable from native execution. For instance, the guest may set up and manage its own paging structures, read and modify control registers, and handle interrupts directly, all of which are tasks that would require the assistance of the hypervisor in more traditional virtualization techniques.

The main difference between VMX non-root and native operation is that interrupts, faults, a task switch, or certain instructions like *cpuid* can cause so-called *VM exits*, which transfer control from the guest to the hypervisor. Additionally, VM exits implicitly save information about the processor's state, and the precise exit reason in a special memory region called the *Virtual Machine Control Structure (VMCS)*. Using this information, the hypervisor can handle VM exits similarly to how a classical operating system would handle an interrupt. Once this process is done, it transfers control back to the guest with a so-called *VM entry*.

Which events precisely should cause a VM exit is highly configurable. A hypervisor may, for example, decide to handle the guest's page faults, while letting the guest handle general protection faults on its own without interference. Guests can also use the *vmcall* and *vmfunc* instructions to trigger a VM exit on purpose, allowing hypervisors to implement a hypercall API for tasks such as communication with other guests.

To further improve performance, *vmcall* and *vmfunc* are sometimes used to mount PV on top of VMX, a technique called *PVH*, or *PVHVM* [25]. However, in contrast to classical PV, a PVH hypercall API is typically strictly required for correct execution, allowing for the virtualization of microkernels without PV support such as Minix [26] or Fuchsia OS's Zircon kernel [27].

Although hardware-assisted virtualization is not necessarily faster than PV [28], it far surpasses PV in popularity today. This is largely because PV requires a much higher implementation effort for both the hypervisor and guest kernel, thus making it unpopular

with developers. It is also far more limited in its capabilities compared to the highly configurable environment created by VMX. Therefore, modern hypervisors for x86_64 rely almost exclusively on hardware-assisted virtualization. A notable exception to this is Xen, which supports PV, hardware-assisted, and PVH virtualization [25, 29].

2.4 Extended Page Tables

An important challenge of creating a fully virtualized environment with VMX is the management of virtual memory. Guests expect to be able to manage their own page tables, which normally requires access to physical memory. However, this stands in conflict with the isolation goals of virtual machines, which makes it impossible to create these page tables without assistance from the hypervisor.
VMX and SVM solve this problem with a mechanism called *Second Layer Address Translation (SLAT)*, which introduces an additional address translation step for virtual machines. When a memory access is performed with SLAT, guest-managed page tables are used to translate the virtual address to a so-called *guest-physical address*. This address is indistinguishable from a physical address to the guest kernel, and can be used in the same way. Meanwhile, the hypervisor manages a second set of page tables for translating the guest-physical address to a hardware-backed *host-physical address*. A memory access therefore involves two address translations, once from guest-virtual to guest-physical and once from guest-physical to host-physical.
On Intel platforms, the hypervisor's second set of page tables are called *Extended Page Tables (EPT)* [9]. What is notable about EPT is that page table entries follow a different format than with the more conventional 4- and 5-layer page tables, which allows for the creation of XOM mappings. An analogous mechanism, called *Nested Paging*, exists for AMD processors [19]. While Nested Paging does not support XOM directly, AMD's SEV-SNP extensions allow for the creation of XOM through a functionally equivalent feature called Reverse Map Tables (RMP).
The primary advantage of using these page-table-based XOM schemes are the significantly stronger security guarantees when compared to MPK. Whereas an unprivileged adversary can disable MPK with just a single instruction, they would have to breach two privilege boundaries to do the same with EPT. It is therefore not necessary to use complex sandboxing techniques to guarantee security. However, this comes at a performance cost. Allocating EPT-based XOM requires not only a system call, but also a hypercall, which makes allocation of such memory regions much slower than conventional allocations. Also, although XOM instruction fetches are not slower than other memory accesses in a virtual machine, virtualization adds a performance overhead by itself, making the technique undesirable for some performance critical applications. The results of this

thesis also suggest that code address translations for EPT-enforced XOM are slower, which may result in runtime overhead for applications that span large numbers of code pages (see Section 5.1.1).

Chapter 3

Key Locking

EPT-enforced XOM is an exceptionally strong security mechanism. As it is managed by the hypervisor, even guest kernels cannot directly disclose its contents. An adversary can therefore not access data stored within such memory regions, even when the guest is fully compromised. This is a security guarantee typically only provided by hardware TEEs, a feature mostly restricted to enterprise-focused CPUs in the context of x86_64. On consumer-level CPUs without TEE support, leveraging the strong security guarantees of EPT-XOM to create protection schemes similar to TEE-enforced confidential computing is therefore a highly interesting prospect.

This chapter proposes *XOM-based Key Locking*, a set of hypervisor-based protection measures and programming techniques that allow for confidential cryptography without relying on a TEE at all. Encryption keys with Key Locking are stored in XOM-protected code rather than data, which allows guests to use them through code execution, but prevents direct read accesses. Due to EPT-XOM's strong security guarantees, attackers cannot disclose these protected encryption keys, even if they manage to fully compromise the guest kernel. At the same time, the code sections containing the keys are executed in the context of a regular user-mode process, requiring no expensive context switches or inter-domain communication to perform confidential cryptography tasks. Also, as the guest does not have direct access to the key, the hypervisor can enforce fine-grained policies on which keys a guest can use at which time, and can revoke the permission to use specific keys even after they were made accessible to the guest.

At its core, Key Locking works by encoding encryption keys as immediate values for *mov* instructions. See Figure 3.1 for a simple illustration of this idea. After this code is initialized, usually at runtime, it is marked as XOM, which prevents the guest from accessing the key again. If the guest is compromised at a later point, the key is therefore protected from disclosure. Alternatively, the code can also be initialized by the hypervisor, so that the guest never has direct access to the key, but can use it regardless.

```
1  .data
2  plain_text: .long 0xcafebabe
3  cipher_text: .long 0x0
4  key: .long 0xdeadbeef
5
6  .text
7  encrypt:
8      // Load key and plain text
9      mov plain_text(%rip), %edi
10     mov key(%rip), %esi
11
12     // "Encrypt"
13     xor %edi, %esi
14
15     // Store cipher text to memory
16     mov %esi, cipher_text(%rip)
17
18     ret
```

(a) Normal implementation

```
.data
plain_text: .long 0xcafebabe
cipher_text: .long 0x0
// Key is now stored as code

.text
encrypt:
    // Load key and plain text
    mov plain_text(%rip), %edi
    mov $0xdeadbeef, %esi

    // "Encrypt"
    xor %edi, %esi

    // Store cipher text to memory
    mov %esi, cipher_text(%rip)

    ret
```

(b) With key in code instead of data

Figure 3.1: Illustration of Key Locking with a XOR-based toy cipher (AT&T syntax).

Unfortunately, XOM by itself is not enough to reliably protect encryption keys from privileged attackers. For example, if the guest kernel interrupts the code in Figure 3.1 after the instruction in line 10 is executed, it can simply read the key from the *esi* register. To prevent this kind of leakage, the hypervisor must overwrite the registers when an interrupt occurs in a XOM page. This approach is called *Register Clearing*.
Additionally, implementations of cryptographic algorithms with Key Locking have to follow a strict set of rules to prevent unintended disclosure of key material. For instance, they cannot write confidential information to regular memory, as this would make it trivially accessible to the guest. While this is easy with the toy cipher in Figure 3.1, it can become quite challenging for real-world ciphers. Also, they have to ensure that their control flow cannot be easily manipulated by a privileged attacker, which among other things, means that they cannot use indirect branches at all.
The following sections describe each of these sub-techniques in more detail. Evaluations of the implementation challenges and performance implications when using Key Locking with real ciphers follow in Chapter 4 and 5.

3.1 Page Locking and Register Clearing

Apart from side-channel attacks and control-flow manipulation, there are two potential ways in which guests could disclose secrets that are stored in XOM: They can either obtain read permissions to the XOM region or interrupt the program while secrets are in use and then disclose them from the register state. Therefore, the hypervisor must

ensure that neither of the two can occur under any circumstance.
The solutions I propose for this problem are *Page Locking* and *Register Clearing*. Page Locking simply means that a XOM page is forcefully overwritten before it is made readable to the guest again. This ensures that its contents cannot be disclosed, while also allowing the guest to free and reuse XOM pages after it is done using them. The only other alternative is keeping the pages as XOM indefinitely, which would eventually drain the guest of its usable memory when allocating new XOM.
Register Clearing means that the hypervisor overwrites the guest's registers before transferring control back to the guest when an interrupt occurs while executing a XOM page. Which registers precisely it must overwrite depends on what the program considers to be confidential: For example, if control flow structures should be kept secret, it may be necessary to perform *Full Register Clearing*. This causes the hypervisor to overwrite all registers, including the instruction pointer and the conditional flags. However, for many applications, especially in cryptography, the control flow is well known and not affected by e.g., the encryption key. In certain cases, it can hence also suffice to clear only a part of the register state.
Most experiments discussed in this thesis therefore employ *Vector Register Clearing*, in which the hypervisor overwrites only the AVX registers and two general-purpose registers. This makes it much easier to recover from clearing events, as the instruction pointer and most general-purpose registers remain unaffected. Of the two general-purpose registers that are overwritten, one is required for transferring data from immediate values in code to the vector registers. The other register serves as a *signal register*, which the hypervisor fills with a magic value during register clearing. By checking the value in the signal register at regular intervals, the program can determine whether register clearing took place, and initiate recovery procedures when needed.
However, Register Clearing also comes at a significant disadvantage: It requires that programs expect clearing events to occur, and can recover from them gracefully. It can hence only be applied to software that is specifically designed with Register Clearing in mind. Note that this is not a problem for Page Locking, as most software does not expect freed memory pages to retain their contents. This makes it possible to run complex applications in page-locked XOM with few modifications. Therefore, there are many scenarios in which programs may want to use Page Locking, but not Register Clearing. To allow for this without endangering confidentiality, the hypervisor treats XOM pages as a simple state machine with three transitions that guests can initiate: *Lock*, *unlock* and *mark*. See Figure 3.2 for a schematic of this. *Lock* transforms a normal memory page into XOM, whereas *unlock* overwrites the page and makes it readable again. A XOM page can additionally be *marked*, which enables Register Clearing. Note however that unmarking a marked page is not possible, as this would make the security guarantees of Register Clearing easily circumventable. Guests must instead use the *unlock* operation to

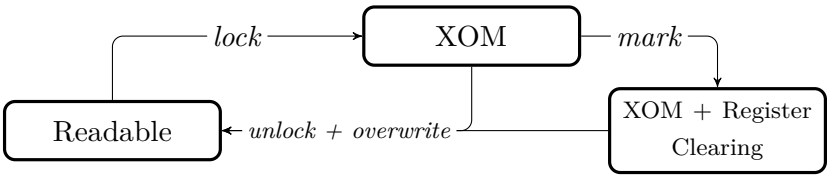

Figure 3.2: State transitions of an EPT-XOM page.

disable Register Clearing, thereby also overwriting the page. This way, the "protection level" of a memory page can only increase, with operations reducing it destroying the secrets stored within.

3.2 Registers as Memory and Encrypted Backups

To ensure confidentiality, no secret data whatsoever can be written to non-XOM memory, as this would always make it accessible to the guest. Confidential information such as encryption keys are therefore kept in the register state at all times. While the capacity of general-purpose registers is usually not large enough to store secrets and perform any meaningful computation at the same time, the vector registers provide significantly more storage space [30]. With AVX2, this gives programs 512 bytes of confidential "memory" to work with, which is ample for many cryptographic algorithms. AVX512 extends this capacity to 2048 bytes, making it feasible to perform more involved computations. The practicality of this technique was already demonstrated by previous works, which primarily utilize it to protect encryption keys against cold-boot attacks (see Section 7.4). What is unique about this work's Key Locking approach is the Register Clearing mechanism. As mentioned in the previous section, Vector Register Clearing ensures that the kernel cannot trivially access the AVX registers through interrupts, thereby preventing the disclosure of cryptographic material. However, this also entails that all progress of a computation is lost when an interrupt occurs. While this is not a problem for programs with a short runtime, an interrupt becomes increasingly likely as the runtime grows larger. With large runtimes, or in environments where high contention for CPU resources causes frequent interrupts, this makes it virtually impossible to run XOM-protected programs with Register Clearing to completion. Therefore, XOM-protected programs that keep secrets in the register state must also have some means to save their progress to regular memory without endangering confidentiality. Data integrity might also be of importance, depending on the application.

For lack of more efficient methods, programs must solve these issues with cryptography. Fortunately, strong encryption is made easy by the AES-NI extensions, which can efficiently encrypt AVX registers with little added complexity [31]. Using AES-NI, AES can even be implemented such that the upper halves of the AVX registers remain untouched,

making it possible to run while preserving 256 bytes of another algorithm's internal state. If integrity guarantees are required on top of confidentiality, programs can additionally employ an authenticated block mode such as the Galois/Counter mode (GCM), which is relatively easy to implement with AVX [32]. Using these methods, programs can securely store any confidential data to memory, allowing them to restore their progress from a backup when register clearing events occur.

The only remaining challenge in this context is to generate a cryptographically random Initialization Vector (IV), which is a parameter in most block modes for AES. If the IV is predictable, or reused for more than one encryption, many block modes become prone to powerful attacks like keystream reuse [33], some of which can break confidentiality with little effort.

There are two possible ways to solve this problem: Programs can either store a confidential seed in XOM for usage with a cryptographically secure pseudorandom number generator (CSPRNG), or if available, use the processor's hardware random number facilities. Out of the two, the latter is usually more practical, as the former method requires programs to keep track of the CSPRNG's state to ensure that the same IV is not used twice. This is difficult, as storing it in the registers may cause it to be overwritten, and storing it in non-XOM memory may allow for external manipulation. Where supported, XOM-protected programs should therefore use hardware facilities instead of software for creating the IV.

3.3 Protecting Control Flow

While Register Clearing prevents guests form trivially disclosing the register state through interrupts, there are other means to leak this data. If an adversary can somehow hijack the control flow, they could redirect it to a disclosure primitive. In the context of Key Locking, preventing this is challenging, as we must assume the adversaries to have kernel privileges, or at least the capability to execute arbitrary code in the same process. We cannot rely on the integrity of non-XOM memory for correct control flow, as these memory regions are under the complete control of the guest kernel, and can be modified in arbitrary ways.

As a consequence, programs cannot use indirect branches that load their targets from memory, such as *ret* instructions. The target address may have been tampered with by either the kernel or a parallel thread, thus allowing an adversary to redirect control flow to a chosen address. However, this does not mean that function-like structures cannot be used at all. See Figure 3.3 for examples of how to create such structures without using memory. Unfortunately, the only reliable workaround is to abandon the concept of a return address altogether, replacing the return instruction with direct jumps. This

```
bar:
    // ...
    test %r8, %r8
    jz .Lreturn_from_bar1
    jmp .Lreturn_from_bar2

foo1:
    mov $0, %r8
    jmp bar
.Lreturn_from_bar1:
    // ...

foo2:
    mov $1, %r8
    jmp bar
.Lreturn_from_bar2:
    // ...
```

(a) Only direct jumps

```
bar:
    // ...
    jmp *%r8



foo1:
    lea .Lreturn_from_bar1(%rip), %r8
    jmp bar
.Lreturn_from_bar1:
    // ...

foo2:
    lea .Lreturn_from_bar2(%rip), %r8
    jmp bar
.Lreturn_from_bar2:
    // ...
```

(b) Return address in register

Figure 3.3: Examples of how to perform a function call without accessing memory (AT&T syntax). Only option **(a)** is reliably secure, as option **(b)** can make the program vulnarable to Spectre v2.

option does not scale well with the amount of possible return targets, as every possible return target requires its own direct jump. In cases where the amount of return targets is large, it might therefore seem tempting to still use a return address, but to store it in a register instead of memory.

However, this bears another risk: As returning still requires an indirect branch instruction, the program becomes vulnerable to Spectre v2, potentially allowing an attacker to redirect speculative control flow to a disclosure gadget. Using established mitigations like retpoline to prevent this is not an option, as retpoline depends on a return instruction [34]. Indirect branches of any kind should thus be avoided completely. Note that while direct branches may still be vulnerable to Spectre v1, programs can ensure that all possible branch targets lie within the XOM-protected program. Defensive programming can thus prevent speculative control flow redirection to a disclosure gadget. With Spectre v2, this is not possible, as the attacker can freely choose the speculative branch target.

Another opportunity for leaking the register state arises when the program crosses a code page boundary. Although the guest kernel cannot access EPT-XOM pages directly, it still has absolute freedom in where to map them in a process's virtual address space. Therefore, if a XOM-protected program loads secrets into the register state, and then crosses a code page boundary, the next page can be any page chosen by the guest kernel. The kernel can thus disclose the register state by inserting its own code there. Unfortunately, this limits the maximum size of a XOM-protected program to just a single code page, unless secrets are encrypted or otherwise protected from disclosure. For complex programs, it may therefore be necessary to use 2 MB pages instead of the more

conventional 4 kB pages.
In summary, a program in EPT-XOM aiming to protect its control flow from privileged attackers must adhere to the following rules:

- Programs must expect every value read from non-XOM memory to have been tampered with, and can thus not rely on them for correct control flow.
- The program cannot cross code page boundaries with secrets in the register state.
- Indirect branches cannot be used in any capacity.
- Programs must be adequately hardened against Spectre attacks, meaning that speculative misprediction of branch targets must not allow for leakage of secrets.

Along with the restrictions on what programs can write to non-XOM memory, following these rules constitutes the greatest challenge when implementing Key Locking for real-world algorithms.
However, there is one additional avenue for redirecting control flow, which they do not cover: A privileged attacker could remap a different code page to the XOM page's virtual address while it is being executed, and the registers contain confidential information. On most processors, this does not endanger confidentiality, because the L1 iTLB typically retains its entries until it is either flushed, or its entries are evicted [9]. With the XOM-protected program covering only a single code page, this cannot occur unless the program is interrupted, and the secrets are overwritten. Therefore, the change cannot become effective while secrets are being handled.
However, note that this is an architecture-specific property, and there are no guarantees that this assumption holds for every processor. It may for example be possible to evict entries from the L1 iTLB through Simultaneous Multithreading, with a potential victim sharing this cache with an attacker thread. The security of Key Locking therefore depends on the assumption that the L1 iTLB is not shared between hyperthreads, and thus cannot be externally modified without an interrupt. While this assumption generally holds for recent Intel processors [35], this is not the case with some processors by AMD[36]. Depending on the underlying hardware, it may therefore still be possible to hijack the control flow of a XOM-protected program, even if it follows all of the above rules. Note however that this attack vector is only available to privileged attackers, and Key Locking may still be useful as a defensive measure against intra-process attacks on affected hardware.

Chapter 4

Implementation

Of the aforementioned methods for creating execute-only mappings on x86-based architectures, only MPK is supported by a commercially used operating system, with Linux making it available through the *pkey_mprotect* system call [37]. EPT-based XOM, on the other hand, was only implemented for research purposes in the past. It is hence not supported by any major hypervisor or operating system, and none of the research implementations [3, 5, 7, 38] are publicly available.
Therefore, this work utilizes a custom implementation of EPT-XOM, which consists of a series of patches to the Xen hypervisor [29], a Linux kernel module called *modxom*, and a user-mode library named *libxom*. See Figure 4.1 for a schematic overview of how these components interact. When all is set up correctly, libxom allows user-mode programs to allocate and manage EPT-XOM through a simple API, and even to migrate their own code into XOM at runtime without endangering stability. To allow for Key Locking, this implementation also provides Page Locking and Register Clearing. The following section describes the overall design and implementation challenges of this system, and how said challenges are solved.

4.1 Modifications to the Xen Hypervisor

4.1.1 Hypercall Interface

To adequately implement EPT-XOM allocation and Key Locking, the hypervisor must make the *lock*, *mark* and *unlock* operations discussed in Section 3.1 available to guests. The easiest way to implement this functionality is through a hypercall, which the guest can explicitly invoke when hypervisor privileges are required. Therefore, the patches for Xen implement them as an extension of the *mmuext_op* hypercall. As a part of

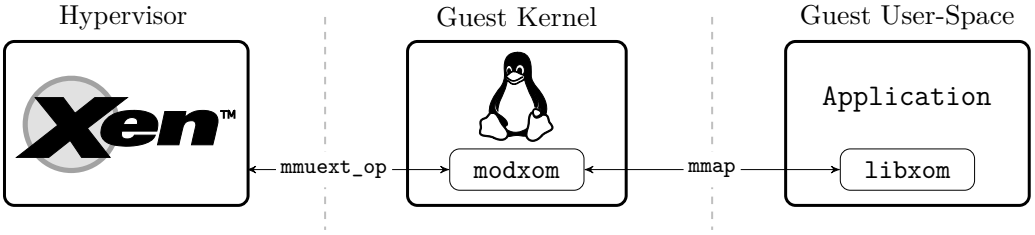

Figure 4.1: A schematic overview of the individual components that make up the EPT-XOM implementation.

Xen's para-API, this hypercall typically provides MMU-related functionality, such as flushing the TLB or installing a new page-table root address in the CR3 control register. However, since hardware-assisted virtual machines can usually perform these tasks without explicitly requesting the hypervisor's assistance, it returns an error code when invoked by such a guest.
For the EPT-XOM implementation, this behavior was modified so that the hypercall can also serve VMX guests. To perform any of the above operations, the guest kernel can then simply issue a *mmuext_op* hypercall, and pass the guest-physical address of a physical page to perform the operation on. Alternatively, the guest can also specify a range of multiple guest-physically contiguous pages, which greatly reduces the number of hypercalls that the guest needs to perform when managing larger XOM ranges.

4.1.2 Register Clearing Modes

The patched Xen hypervisor supports both Full Register Clearing and Vector Register Clearing. Guests can specify the desired Register Clearing mode when issuing *mark* operations, but changing the Register Clearing mode once a page is marked is no longer possible. When the hypervisor handles a VM exit, it translates the guest-virtual address in the *rip* register to a guest-physical address using the guest's paging structures and checks whether the resulting page frame number corresponds to a previously marked XOM page. Depending on whether or not this is the case, Xen then zeroes the guest registers according to the clearing mode. The general purpose registers overwritten with Vector Register Clearing are *r14* and *r15*, with *r15* being the signal register.
Note that for Register Clearing to be reliably secure, Xen must actively handle all of the guest's interrupts and faults, including those that the guest could normally handle by itself. A potential threat to security in this context is the virtual interrupt controller (vAPIC) that many processors with hardware virtualization support [9]. This hardware facility allows guests e.g., to perform inter-processor interrupts by themselves, which would otherwise require the hypervisor's assistance. What is problematic about the vAPIC is that it delivers interrupts directly to the guest kernel, giving the hypervisor no

chance to enforce Register Clearing policies. For this reason, the patched Xen hypervisor disables the vAPIC by default, which guarantees security, but incurs a slight performance penalty on guests (see Section 5.1.3 for a more comprehensive elaboration on this).

4.1.3 Handling EPT Violations

Any violation of the permissions specified in the EPT causes an immediate VM exit. By retrieving the exit reason from the VMCS, Xen can then determine that the exit was caused by an EPT violation, and take appropriate action.
In cases where the EPT violation is caused by a read or write access to XOM, the modified Xen hypervisor simply injects a general protection fault into the guest. This type of exception also occurs for unauthorized accesses to memory protected by the conventional 4- and 5-level page tables, and most guest kernels thus handle it in the same way. On a Unix-like operating system, this results in the delivery of a segmentation fault signal to the offending process. Processes can handle this gracefully by setting up a signal handler, but crash otherwise.

4.2 Linux Kernel Module

To make EPT-based XOM available to user-mode programs, the guest kernel must provide a user-mode interface to the hypervisor's newly added functionality. While creating this interface may seem trivial on the surface, allocating memory areas that not even the kernel can access introduces some issues, which may endanger the kernel's stability. For example, a XOM page cannot be swapped out to disk, and any attempts to do so lead to error conditions that the Linux kernel cannot recover from. Similar issues arise when a process using XOM pages terminates without freeing them first, and the kernel reassigns these pages to a different process without invoking the hypervisor first. Unfortunately, there are many such scenarios in which the Linux kernel may attempt to read from a XOM page. Some can even be triggered by the user through system calls, enabling attackers to perform denial-of-service attacks on an improperly implemented XOM interface. Reviewing and modifying all of them would require substantial development effort, and is hence out of scope for this thesis. For this reason, the EPT-XOM implementation may lead to unexpected crashes and should not be used on production systems.
However, many of the aforementioned problems can be mitigated by implementing a separate in-kernel memory allocator, and restricting XOM to memory allocated this way.

Such a system has several advantages:

- A user-supplied code page may already be mapped into the address space of multiple processes, potentially even of different users. Were we to lock such a page, these other processes may crash as a result. However, this cannot occur when we force the user to allocate pages specifically for XOM. Note that a XOM page may still be used in multiple address spaces this way, but only in cases where a parent process allocated it as a XOM page before forking.
- The allocator can implement the *mmap* system call in conjunction with a file descriptor. Should a program using XOM pages terminate for any reason, including through a crash, we receive a notification when the file descriptor is closed. This allows us to reliably release any XOM pages that are still in use at this point.
- We have full control over the memory backing. For example, the kernel could crash if a XOM memory range were backed by a Unix file object instead of guest-physical memory. This is because, depending on how the underlying mechanism of the file is implemented, locking its physical memory likely can lead to unauthorized read and write accesses in the kernel. If we were to allow locking arbitrary memory supplied by the user, such cases would need to be taken into consideration.
- Because of the above, we can safely use mechanisms that prevent XOM pages from being swapped out to disk.

modxom, the kernel module providing EPT-XOM functionality to user-mode applications, hence implements its own memory allocator. User-mode programs can use this allocator by first opening the `/proc/xom` file that modxom creates during initialization, and then using the file descriptor with *mmap*. Operations like locking or freeing XOM pages are then performed by writing a command struct to the file descriptor. See Figure 4.2 for an example of this workflow.
Unlocking pages works analogously with a different command code in `modxom_cmd::cmd`. Note that this does not unmap unlocked (and thus overwritten) pages from the address space, which requires a separate call to *munmap*. At the same time, *munmap* does not unlock the page if it is still locked. Because modxom's in-kernel memory allocator keeps the page allocated in this state until the process terminates, unmapping locked XOM pages can lead to memory leaks. These memory leaks are hard to detect, as the improperly unmapped pages are associated with the kernel instead of a user-mode process. Programs using modxom must therefore make sure to always unlock XOM pages before unmapping them. Closing the file descriptor or terminating the process results in modxom immediately unlocking the process's remaining XOM pages, including those that were unmapped while locked.

```
1  // Open the XOM file
2  int xomfd = open("/proc/xom", O_RDWR);
3
4  // Allocate a (still writable) XOM page
5  char* xom_page = mmap(NULL,
6      1 << 12,
7      PROT_READ | PROT_WRITE,
8      MAP_PRIVATE,
9      xomfd, 0);
10
11 // Initialize the page with code
12 memcpy(xom_page, code_pointer,
13     code_size);
```

(a) Allocation of a XOM page

```
1  // Initialize command struct
2  struct modxom_cmd cmd = {
3      .cmd = MODXOM_CMD_LOCK,
4      .num_pages = 1,
5      .base_addr = xom_page,
6  };
7
8  // Lock the page
9  write(xomfd,&cmd,sizeof(cmd));
10
11 // Reads/writes now cause a #GP fault
12 char v = xom_page[0];
13 // Unreachable
```

(b) Locking a XOM page

Figure 4.2: Using modxom in a user-mode program on Linux

4.3 User-Mode Library

While providing a system call-based user mode interface to manage page-locking is technically sufficient for mounting experiments, using these system calls directly becomes unwieldy with more complex tasks. As part of this thesis, I therefore present *libxom*, a small Linux user mode library serving as an easy-to-use interface for page-locking. If EPT-enforced XOM is not supported, *libxom* can also emulate Page Locking behavior with MPK for testing. In summary, the libxom library provides the following features:

- Safe de/allocation of EPT-enforced XOM through a unified interface
- Simplified handling of sub-page XOM
- Functions for migrating a process's code into XOM at runtime
- An easy-to-use primitive for recovering from Register Clearing events

Furthermore, libxom is accompanied by the `xom` command line utility, which can launch any process with its entire code in XOM.
At the core of libxom is the concept of a XOM buffer, which is backed by either MPK or EPT-based XOM. By default, libxom chooses EPT over MPK if modxom and modified Xen are available, but the user can explicitly request MPK-enforced XOM if so desired. The XOM buffer itself is represented as a pointer to an anonymous struct. To write into the XOM buffer, lock it or mark it for Register Clearing, the application can use this pointer as a parameter in libxom's API. A pointer to the XOM pages themselves is only returned by the *lock* function. This way, there is no risk of the user calling a function in XOM before locking it. Allocating and using sub-page XOM buffers works analogously, but utilizes a separate set of API functions. To illustrate how XOM buffers can be used, Appendix A contains the code for a simple demo program, which allocates and utilizes a

XOM buffer with a range of libxom's functionalities.

Furthermore, libxom can migrate a process's existing code into XOM through the *xom_migrate_all_code* function. When called, this function temporarily remaps all executable memory ranges to a different location in the virtual address space. It then allocates XOM in the code's original location, copies the code into it, and locks it before resuming normal execution. Once this is complete, all memory in the program's address space is either readable or executable, but never both. All instances of the original code are unmapped or overwritten, thereby making it fully inaccessible to reads. The only exception to this is the processes vDSO section [39], the correct usage of which necessitates introspection. By declaring a *constructor function* [40], which is called when libxom loads and thus executes before *main*, libxom can even perform the migration before a program's own code is first invoked. As part of this procedure, libxom also installs hooks for the *dlopen* and *dlmopen* functions to migrate libraries loaded at runtime into XOM without necessitating code changes. If it is requested in the environment variables, a program using libxom can therefore launch with its entire code in XOM.

Using this feature, the `xom` utility accompanying libxom can launch any dynamically linked ELF executable in XOM from the command line, without having to modify any code. If the target executable does not link to libxom itself, `xom` forces glibc's dynamic linker [41] to load it into the address space by configuring the *LD_PRELOAD* environment variable. Unless the environment variables are actively modified, this also extends to all of the target executable's child processes, making it possible to run even complex applications entirely in XOM.

Note that this usually does not endanger stability, as modern compilers strictly separate code and data for performance reasons [42]. Crashes generally only occur for programs written in assembly languages, where this separation may not be so strict. A notable example of this is OpenSSL, which requires extensive refactoring to run reliably in XOM.

Finally, libxom provides an easy means to recover from Register Clearing events. Using the `expect_full_register_clear` macro, developers can declare a code block that simply repeats if the registers are cleared during execution. See Appendix A for an example.

Internally, the macro creates a non-local jump target with the C standard library's `setjmp` function, thereby saving most of the program's relevant state to memory. Additionally, it sets up a signal handler for segmentation faults, which are triggered when a program attempts to fetch code from the NULL page after a Register Clearing event. From there, the program performs a non-local jump to the previously created target, thus restarting the block.

An implication of this system is that such a block can only terminate when it completes without an interrupt occurring. Applications must therefore ensure not to accidentally prevent this from happening. If possible, longer-running tasks should periodically save

their progress to memory, so that progress is not lost when an interrupt occurs. Also note that any system call from within a XOM page with Register Clearing has the same effect as an interrupt, potentially causing an infinite loop if no precautions are taken.

4.4 Limitations

Currently, the modified Xen hypervisor only supports page locking on Intel platforms. This is mainly due to differences in the respective SLAT implementations of VMX and SVM, which necessitate platform-specific programming when modifying page table entries. VM exit handlers are also platform-specific, which affects the implementation of Register Clearing. Therefore, reliable page-locking for both Intel and AMD platforms would essentially require two separate implementations.
In the interest of time, only the Intel-specific version was implemented for this thesis. The reason for choosing Intel over AMD is that creating SLAT-based XOM on AMD requires the SEV-SNP feature set extension, which is far less widely supported on consumer-level CPUs than Intel's EPT. Furthermore, Xen already supports a high-level interface for managing EPT entries, whereas support for SEV-SNP is fully absent at the time of writing. To what capacity page locking on AMD platforms differs from page locking on Intel platforms therefore remains a question for further research.

4.5 Availability

All code written as part of this work is made publicly available in hopes that it may be useful for future research. This includes the modified Xen hypervisor, modxom, libxom, and most of the experiments. To ease the process of setting up a XOM-capable environment, I also provide a disk image of a fully set up Xen hypervisor with a host and XOM-capable guest operating system.
The following is a list of publicly accessible software repositories that were used during the creation of this thesis.

- `https://github.com/tristan-hornetz/libxom` - Contains libxom and modxom. This repository is geared towards public release and may see updates in the future.
- `https://github.com/tristan-hornetz/xen` - The modified Xen hypervisor.
- `https://github.com/tristan-hornetz/libxom-experiments` - An archive of the code used for benchmarks and attack proof-of-concepts. This repository also

contains the exact versions of modxom and libxom that were used during these experiments to ensure reproducibility.

- `https://github.com/tristan-hornetz/openssl` - A patched version of OpenSSL, where code and data are more strictly separated. It is meant for usage with nginx.

Chapter 5

Evaluation

As stated earlier, the primary goal of this thesis is to assess the characteristics of hardware-enforced XOM on a more fundamental level, in order to identify novel applications for it. Therefore, this chapter contains a survey of XOM's performance implications and an analysis of multiple attacks that might threaten its security guarantees. Since Key Locking is perhaps the most important result of this effort, this chapter also contains case studies of Key Locking implementations for AES-128-CTR and HMAC-SHA256, covering their specific implementation challenges and performance as compared to a non-confidential reference implementation.

5.1 Performance

Many potential applications of XOM hinge on the assumption that XOM-protected code imposes little overhead on a program's execution. If used as a defense against code-reuse attacks, for example, the security benefits of a XOM-based protection scheme may not warrant any overhead at all in the eyes of many users. This especially holds in highly optimized environments, such as a web browser or a database. While existing protection schemes involving XOM are known to slow the protected program down, it is unclear to what capacity XOM itself contributes to this overhead, as these schemes also involve drastic changes to a program's code [3, 8].
Therefore, I investigated the performance of XOM on its own, first on a set of micro-benchmarks, and then on the *nginx* webserver [43] to gauge the effects on a more complex workload. The results show that MPK-enforced XOM is just as fast as regular memory, whereas EPT-enforced XOM can impose a non-negligible runtime overhead, even without modifications to the code.

5.1.1 Microarchitectural Performance

In addition to finding out whether XOM can slow a program down at all, a goal of this performance analysis is to identify the exact circumstances under which this can occur. This is not possible by simply benchmarking a set of test programs such as the SPEC CPU benchmarks [44], as these benchmarks typically consist of complex workloads that may obscure the microarchitectural effects responsible for potential overhead. Consequently, the first set of tests consists of simple micro-benchmarks, which aim to identify performance differences between XOM and non-XOM memory for primitive microarchitectural operations. Specifically, the goal of these tests is to study the effect of XOM on instruction fetches, code address translations, and speculation behavior. To achieve this, the following micro-benchmarks are employed:

- **Access**: Call a function that consists of a single `ret` instruction, thereby triggering a single-byte memory access. In the **cached** variant, a 'dry-run' of this benchmark is performed before running the test itself, to ensure that the function's code is cached. In the **uncached** variant, the function's code is flushed from the cache hierarchy before it is called with the `clflush` instruction.
- **NOP Sled**: Call a 4 kB page filled with `nop` instructions. The goal of this test is to find out whether there are performance differences in branchless code without complex control-flow and memory dependencies, which does not trigger code address translations. As the CPU does nothing but fetch the code from memory, this should give insights into the performance of memory accesses into XOM.
- **Primes**: Compute the first $10,000$ prime numbers with a naive trial division algorithm. The goal of this test is to find out whether there are performance differences in scenarios where correct code branch speculation is harder, and occasional mis-prediction is to be expected. However, this workload also greatly benefits from correct speculation, as it contains loops that repeat often and exhibit similar behavior for extended periods of time. The *Primes* benchmark is thus likely to perform differently when speculation behavior is altered.
- **Chain**: This test consists of 2^{12} 4 kB code pages. Whenever a page is called, the program simply jumps to the next page until the end of the chain is reached. The number of pages is larger than the amount of entries in the last-level TLB, thus necessitating address translations even when the test is performed repeatedly.
- **Random Jumps**: As with *Chain*, this benchmark consists of 2^{12} code pages. However, instead of jumping along the chain sequentially, the next page is chosen at random with a linear-feedback shift register. This ensures that the next page

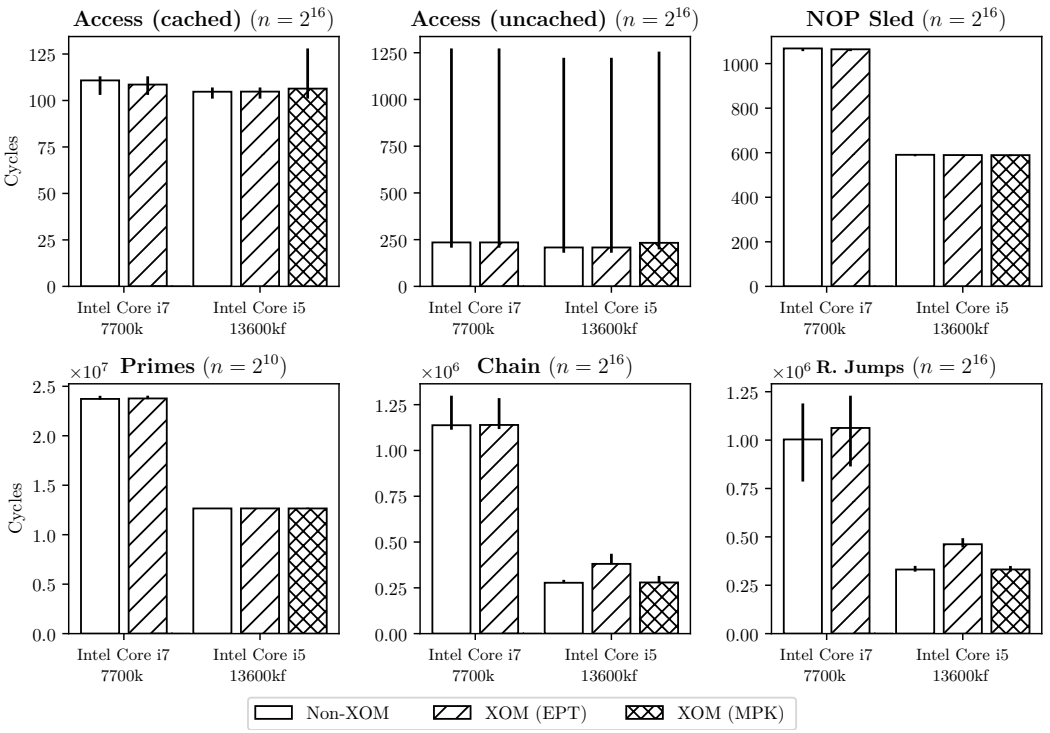

Figure 5.1: Mean time required for executing the benchmarks with and without XOM (less is better). The black lines in the center of the bars indicate the range between the 1st and 99th percentile of samples.

address is hard to predict, thus preventing the CPU from performing address translations ahead-of-time.

Test results for the Intel Core i7 7700k and i5 13600kf processors are shown in Figure 5.1. Note that MPK is not supported on the Core i7 7700k, so times for MPK-enforced XOM are only listed for the Core i5 13600kf.

For the *Access*, *NOP Sled*, and *Primes* tests, there is no significant timing difference between non-XOM code and code protected by either of the XOM variants. This strongly indicates that the performance of instruction fetches is unaffected by XOM, regardless of whether the code is cached or uncached. Speculation remains unaffected as well, as indicated by the *Primes* benchmark.

However, there is a large difference in the execution times of the *Chain* and *Random Jumps* tests between non-XOM and EPT-XOM. With the *Random Jumps* benchmark, the mean runtime overhead is roughly 5.9 % for the Core i7 7700k, and 39.4 % for the newer Core i5 13700kf. For the *Chain* benchmark, the Core i7 7700k does not exhibit an overhead at all, whereas an overhead of roughly 37.1 % is observable on the Core i5 13700kf. From this, we can confidently conclude that EPT-XOM code address translations are slower on both processors. Note that all benchmarks use EPT, and that the only difference between the non-XOM and EPT-XOM benchmarks is the setting of the EPT permission bits. Therefore, this cannot be explained as a performance difference

between EPT and non-EPT address translations. Notably, there is no overhead with MPK-enforced XOM, which makes sense given that the page table entries of MPK-XOM are not modified when a page is locked.
However, the cause for the large divergence between the two processors remains unknown as of writing this thesis. It is possible that the Core i7 7700k exhibits a more aggressive prefetching strategy, already performing the address translations for the next pages while the previous page's code is still executing. This would explain the lack of a runtime overhead with the *Chain* benchmark, as the next pages are easily predictable. Additionally, the sub-mechanism causing the overhead may not have been optimized as much across processor generations as address translations overall. The Core i5 13700kf can perform a normal address translation significantly faster, thus causing a larger relative overhead if the absolute overhead time remains constant.
A consequence of the EPT-XOM's overhead is that large programs with many cross-page branches are more affected than smaller programs occupying only a few code pages. Whereas simple programs, such as the trial division algorithm in the *Primes* benchmark, may not exhibit any runtime overhead at all, complex applications may see a significant slowdown due to EPT-XOM. Therefore, EPT-XOM is generally better suited for protecting small code segments rather than entire applications, which may make it a poor choice e.g., as a defense mechanism against code-reuse attacks.

5.1.2 Setup Performance

To estimate the impact of XOM on real-world applications, the costs of setting up and releasing XOM pages must also be taken into account. For MPK-XOM, this cost does not differ from allocating normal memory. On Linux for example, MPK-protected pages are allocated and freed using the same mechanisms as unprotected pages. The cost of locking them is negligible, as this only requires a single register access.
Unfortunately, however, allocating and freeing EPT-XOM pages with libxom is considerably slower than allocating and freeing normal 4 kB pages. The average runtime required for performing these tasks is shown in Figure 5.2. For comparison, this figure also includes the average time required with the conventional *mmap*/*munmap* interface. At the time these benchmarks were performed, the test system always had enough memory available to allocate the pages, so potential costs incurred by swapping memory to disk are not included.
Whereas the allocation time for normal pages is almost constant with respect to allocated size, it grows in a roughly linear fashion for XOM pages. This is mostly due to the mechanism that Linux uses to mark pages as non-swappable, which requires memory ranges to be entered on a per-page basis. While this is hard to fix from within a kernel

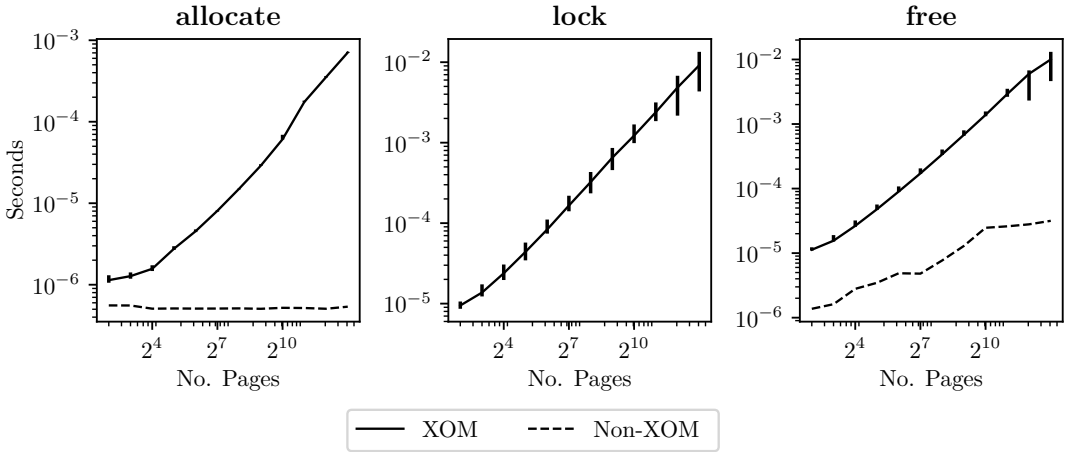

Figure 5.2: Mean time required for allocating, locking, and freeing 4 kB pages ($n = 2^{10}$, Intel Core i5 13600kf). The vertical lines indicate the range between the 1st and 99th percentile of samples.

module, this mechanism could easily be optimized in the kernel itself, so allocating non-swappable memory for use as XOM can in theory be done with little overhead.

Costs for locking a memory range also grow linearly, which is due to similar per-page mechanisms in Xen and modxom. To make locking more efficient, Xen allows locking a range of physically contiguous pages with a single hypercall. Therefore, modxom performs contiguity checks on pages that are to be locked, incurring linear costs. Similarly, as an EPT entry must be modified for every XOM page, Xen must perform a page table walk for every page it locks.

Releasing XOM pages incurs linear costs in Xen for the same reason. Additionally, Xen overwrites the entire memory range upon releasing it. While the Linux kernel does this as well when freeing a page, releasing a XOM page is still orders of magnitude slower, due to the combined costs of hypercalls and having to modify multiple sets of page tables.

However, the impact of this overhead on most real-world applications is still likely to be little. Allocating and releasing code pages is typically only done upon starting and stopping a process, so an impact on the program's runtime performance is unlikely. Furthermore, startup performance is unlikely to be noticeably different to most users, as allocating code pages is still relatively fast, even if slowed down by multiple orders of magnitude (for reference, allocating 16 MB of EPT-XOM takes roughly 700 µs, and locking it takes about 9 ms on an Intel Core i5 13600kf). Also, note that the test implementation used in this thesis leaves much room for optimization. Although performing an additional hypercall from within the kernel is necessarily slower than a normal system call, all of the above operations can likely be performed much faster if sufficiently optimized. This section's results should therefore be seen as an upper bound on setup performance, rather than absolute values.

Nevertheless, there are scenarios in which EPT-XOM can severely degrade performance,

such as environments with a JIT-compiler, where allocating and releasing large segments of code during runtime is not out of the ordinary. If possible, applications like web-browsers should therefore rely on MPK-enforced XOM, which incurs negligible overhead.

5.1.3 Application Performance

To get an understanding of how XOM impacts a more complex application, I performed a set of benchmarks on the nginx webserver using the *Phoronix Test Suite* [43, 45]. Phoronix measures the amount of *https* requests that nginx can process over a fixed period of time, saving the aggregated value as a sample every 90 seconds. This process repeats at least three times and then continues until the standard deviation of samples is below a threshold of 3.5 % of the arithmetic mean. The amount of recorded samples thus varies between configurations, but the results are always guaranteed to be statistically significant. Phoronix can perform this test with different numbers of clients querying nginx in parallel, all officially supported settings of which were tested in this experiment. For the test runs using XOM, nginx was started with libxom's `xom` command line utility, which ensures that all code is migrated into XOM before it can execute (including libraries loaded at runtime with *dlopen*). The effectiveness of this method was verified by externally monitoring nginx's memory map while the tests were performed. Additionally, both XOM and non-XOM tests employed a customized version of OpenSSL, as OpenSSL does not strictly separate code and data in its assembly code and thus requires modifications to execute correctly in XOM.
The results of this benchmark are shown in Figure 5.3. On average across all configurations, EPT-XOM reduces the amount of requests nginx can process by 5.3 % on the Core i7 7700k, and by 3.1 % on the Core i5 13600kf. This confirms that EPT-XOM can incur non-negligible performance overhead on real-world applications, even without diversification or other modifications to a program's code.
In contrast, MPK-enforced XOM leads to a reduction of only 0.6 % on the Core i5 13600kf, which is well within the error expected with Phoronix's 3.5 % standard deviation threshold. Although a small overhead cannot be ruled out, this reduction is likely a result of measurement imprecision, given that MPK-XOM does not produce observable overhead in any of the previous experiments. MPK should therefore be the preferred XOM enforcement mechanism when protecting whole applications, unless the protection scheme depends on EPT's stronger security guarantees.
In this context, it should be noted that there is a second type of overhead that can impact applications if Register Clearing is in use. For the Register Clearing mechanism to be secure, the hypervisor must intercept all of the guest's interrupts, faults, and

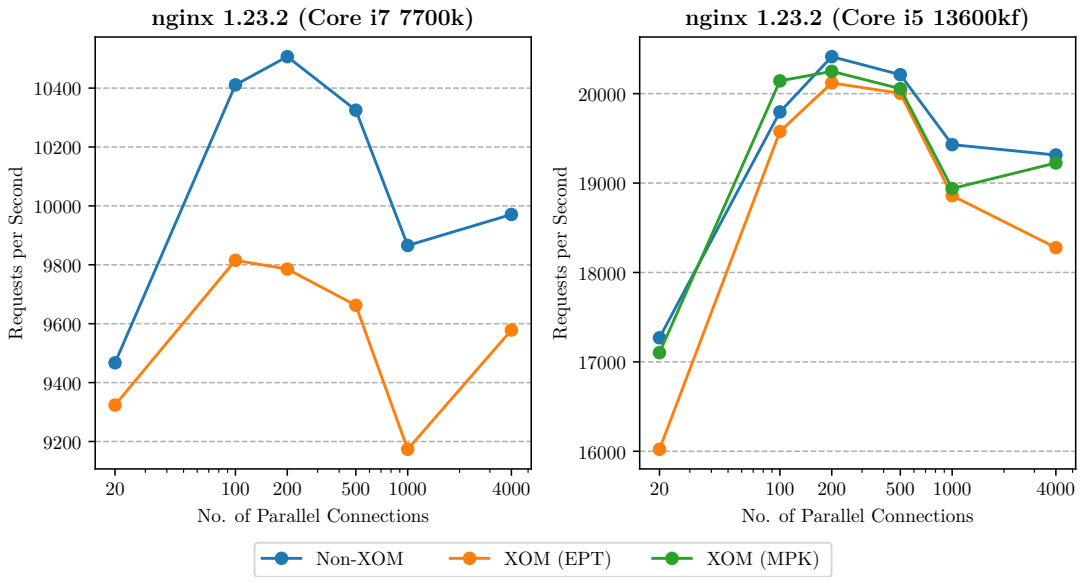

Figure 5.3: Average performance of a 2-threaded nginx instance with and without XOM, measured in requests processed per second (more is better).

other task switches, even those that the guest could normally handle without external intervention. This also entails that the vAPIC included in VMX must be disabled, as it delivers interrupts directly to the guest kernel [9]. The performance penalty resulting from this affects the entire guest system, even if EPT-XOM or Register Clearing are not used directly.

For the first test, register clearing was completely removed from Xen, whereas the second test enables register clearing and disables the virtual APIC. XOM was not utilized in any of the configurations.

On average, register clearing reduces the amount of requests nginx can handle by 1.8 % on the Core i7 7700k, and by 2.2 % on the Core i5 13600kf. However, these values should be seen as an upper bound for the expected performance overhead in the same way as the results from Section 5.1.2. Firstly, the register clearing implementation used for this experiment is not particularly optimized, and can likely still be improved upon significantly. While an overhead of roughly 2 % is unacceptable in many scenarios, the overhead with a well-optimized implementation may therefore be much lower. Secondly, a web server attempting to process as many requests as possible constitutes somewhat of a worst-case scenario for register clearing, as this involves a high amount of task switches, which need to be handled by the hypervisor. More compute-heavy applications with fewer task switches may therefore see a lower performance penalty.

Despite its system-wide costs, register clearing is thus not impractical. In comparison to certain Spectre mitigations, which can slow down nginx by as much as 25 % [46], these costs even seem downright negligible. Nevertheless, register clearing should only be enabled if absolutely required to prevent unnecessary costs. Figure 5.4 shows the effect

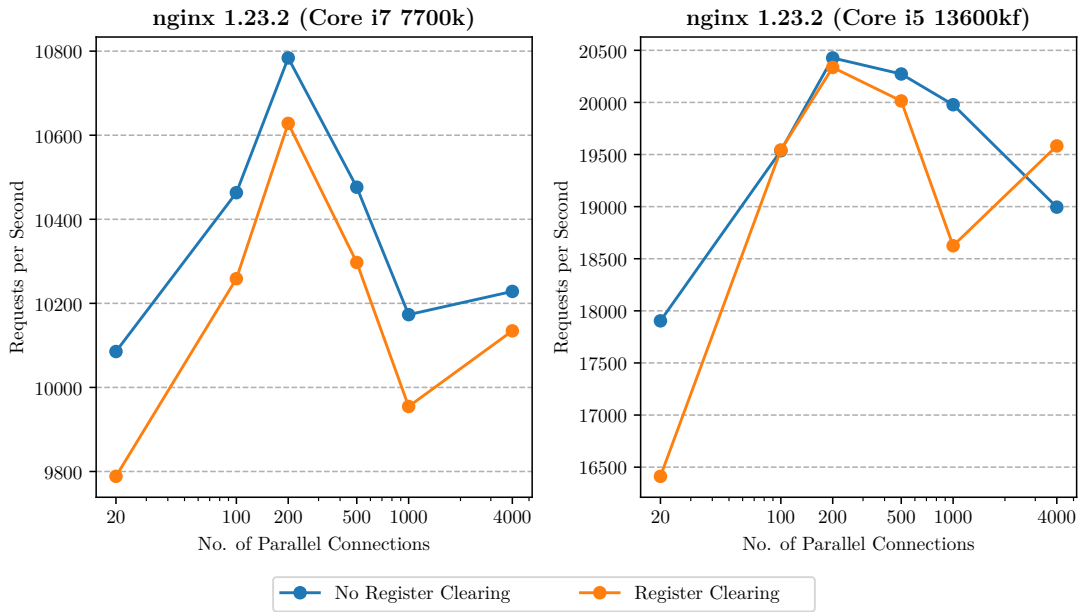

Figure 5.4: Average performance of a 2-threaded nginx instance with and without Register Clearing enabled in Xen, measured in requests processed per second (more is better). None of the configuration utilitze XOM.

this has on the performance of nginx.

5.2 Attacks on Execute-Only Memory

The following section aims to investigate potential attacks against XOM. This only covers attacks applicable to hardware-enforced XOM in general, meaning that they are oblivious to the setting in which it is used. In summary, XOM proves to be highly resilient against transient execution attacks such as Spectre (see Section 5.2.3) and Meltdown (see Section 5.2.4), even to the point that it can be considered a countermeasure against them. On the other hand, attacks like Interrupt-driven Code Recovery can almost completely dismantle the security guarantees of XOM and are difficult to mitigate (see Section 5.2.1). Despite of this, none of the attacks discussed in this section can undermine the confidentiality of Key Locking, reaffirming the technique's effectiveness.

For all attacks, we assume that the protected code is already locked into XOM once the attack begins and that there are no copies of the protected code anywhere in memory. The attacker's goal, unless otherwise specified, is to disclose as much information about the XOM-protected code as possible. No information about the code is known to the attacker before starting the attack, apart from its intended usage.

5.2.1 Interrupt-driven Code Recovery

A special property of EPT-based XOM is that even the guest kernel is unable to read its contents. Therefore, secrets stored in this type of XOM may be the target of attackers with kernel privileges, which means that we have to take such attackers into account when discussing this protection scheme.
Perhaps the most effective attack that a kernel-based attacker can perform is *Interrupt-driven Code Recovery* [1]. First described by Schink and Obermaier for the ARM Cortex-M architecture, this attack involves the following steps for recovering a single instruction:

- Generate a set of so-called *input states*, each of which with different register and memory values.
- For each input state, execute the instruction and interrupt the program right afterward to record the resulting *output state*.
- Filter out implausible instruction types based on a set of simple rules.
- Enumerate all of the remaining instructions.
- Verify these instructions by executing them on the input states, discarding those that do not produce the recorded output state.
- If only one of the enumerated instructions is left, the original instruction was successfully recovered. Otherwise, the attacker is left with a set of functionally equivalent instructions, one of which is guaranteed to be the original.

Note that with debugging interfaces such as *ptrace* on Linux [37], this attack can also be mounted by a user mode process. However, as it is trivial to restrict access to these interfaces, we consider this to be an attack requiring kernel privileges.
While this method of interrupt-driven code recovery is effective for Cortex-M, several issues arise when conducting similar attacks on x86-based architectures. Firstly, x86 is a CISC instruction set, and as such, the amount of instructions it supports is considerably larger. This makes it challenging to filter out instruction types on just a few simple rules. Secondly, the encoding of x86 instructions allows for greater variation in terms of an instruction's operands. For example, memory accesses require a dedicated instruction with the Thumb instruction set used on Cortex-M [47]. On x86_64, most arithmetic and branch instructions can use a memory operand, which may consist of a base register, index register, scale factor, and a 32-bit displacement value [31]. Due to this massively increased set of instruction variants, it is impractical, if not impossible to enumerate a

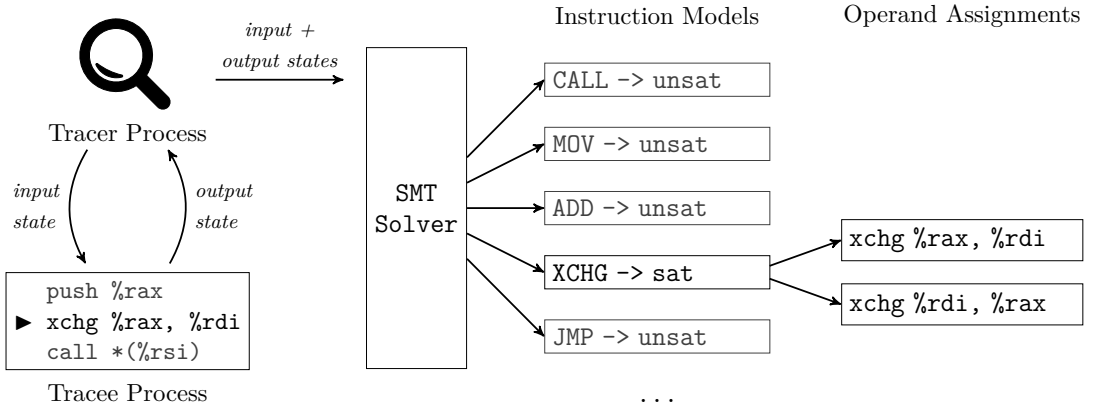

Figure 5.5: A schematic overview of interrupt-driven code recovery on x86_64. The assembly snippets use AT&T syntax.

set of all instructions that come into question for a specific output state. This problem is unsolved as of yet, and to the best of my knowledge, interrupt-driven code recovery has not yet been attempted on x86_64.

As part of this thesis, I therefore propose a modified version of interrupt-driven code recovery for complex instruction sets such as x86_64, with the goal of demonstrating that this attack is a realistic threat. Instead of enumerating all plausible instructions based on simple rules, the instruction list is generated with an SMT solver. A high-level overview of this technique is shown in Figure 5.5. A *tracer process*, which is either a kernel procedure or a user-mode process using *ptrace*, generates a set of input states, lets the so-called *tracee process* execute a XOM instruction on them, and then retrieves the output state. Both the input and output state include the current register values and the values of certain memory locations, such as the current stack frame and all locations to which a pointer is stored in a register.

An SMT Solver then compares these states to a set of so-called *instruction models*. Each instruction model consists of a set of constraints on the input and output states in relation to an instruction's operands, representing the logic of a certain family of instructions. For instance, the model for *ret* contains a constraint dictating that the value of `%rsp` in the output state must have been increased by 8 compared to the input state, and another constraint dictating that the current output state's `%rip` must be the value in the input state's `(%rsp)`. The model for the family of *mov* instructions dictates that if a register has changed in the output state, the destination operand must be said register and the destination's new value must either be a value from another register, a memory location, or the instruction's immediate value. Only if the SMT solver finds an assignment to the instruction's operands such that the model is satisfied, the resulting instruction is considered for verification.

The effectiveness of this altered approach was tested on a small proof-of-concept implementation, based on the z3 SMT solver [48]. Since modeling all instructions of x86_64

```
1  get_fibonacci:
2   movq $0, (%rsi)
3  _1:
4   cmp $1, %rdi
5   jg 0xc <_2>
6   // Base case: n <= 1
7   lea (%rsi,%rdi,8), %rcx
8   mov %rdi, %rax
9   dec %rdi
10  jmp 0x11 <_3>
11 _2:
12  // Recursive step
13  dec %rdi
14  call -0x1a <_1>
15  xchg %rax, %rdi
16  add %rdi, %rax
17  add $8, %rcx
18 _3:
19  // Save result
20  mov %rax, (%rcx)
21  ret
```

(a) Original Code

```
get_fibonacci:
 movq $0, (%rsi)
_1:
 cmp $1, %rdi
 jg 0xc

 lea (%rsi,%rdi,8), %rcx
 mov %rdi, %rax
 <dec %rdi | sub $1, %rdi>
 jmp 0x11
_2:

 <dec %rdi | sub $1, %rdi>
 call -0x1a
 xchg <%rdi,%rax|%rax,%rdi>
 add %rdi, %rax
 add $8, %rcx
_3:

 mov %rax, (%rcx)
 ret
```

(b) Output of z3

```
get_fibonacci:
 movq $0, (%rsi)
_1:
 cmp $1, %rdi
 jg 0xc

 lea (%rsi,%rdi,8), %rcx
 mov %rdi, %rax
 dec %rdi
 jmp 0x11
_2:

 dec %rdi
 call -0x1a
 xchg <%rdi,%rax|%rax,%rdi>
 add %rdi, %rax
 add $8, %rcx
_3:

 mov %rax, (%rcx)
 ret
```

(c) After verification

Figure 5.6: A function filling the buffer in `%rsi` with the first n Fibonacci numbers, where n is given in `%rdi` (AT&T syntax). Figures **(b)** and **(c)** show the formatted output of z3, and the recovered code after verifying each instruction on the input states.

as constraints would require an effort incompatible with the time limits of this thesis, the proof-of-concept only considers a small selection of the most commonly used instructions. The input state generator is also relatively primitive: For each input state, one register or memory value of the state recorded during normal execution is replaced with a constant integer, a valid data address, or a valid code address.

Figure 5.6 shows the output of this implementation for a short program computing the Fibonacci number sequence. Despite the input generator's simple design, z3 can uniquely determine all but three instructions purely based on the recorded output states. Verification on hardware was therefore not necessary with these instructions. Note however that the proof-of-concept implementation's search space is reduced to just a few instruction families, and that other instructions may come into question as well if the whole instruction set were to be considered. For the remaining instructions that could not be uniquely determined, the alternatives are functionally equivalent. However, the alternatives generated for the instructions in lines 9 and 13 have different lengths when assembled, which leaves only one valid instruction after verification. Only the `xchg` instruction in line 15 cannot be uniquely recovered due to its commutative properties. The reconstructed code perfectly reproduces the original code's behavior nonetheless, regardless of which alternative is chosen.

Nevertheless, this type of attack is not without its limitations. As demonstrated, certain instructions with commutative properties such as `xchg` cannot be recovered, even in

full knowledge of all possible input and output states. The same holds for instructions without an observable effect, such as `nop`, `mov %rax, %rax`, or `add $0, %rax`. While this may not be of concern to an attacker aiming to build a functional reconstruction of the XOM-protected code, an attacker aiming to disclose the code as a prerequisite for code-reuse attacks may not be able to recover certain gadgets due to this.

Moreover, this attack is resource-intensive in terms of computing power. To produce the output in Figure 5.6, z3 required a runtime of 22 min 2 s on an Intel Core i5-13600kf processor with 20 parallel threads, evaluating roughly 150 input and output states per instruction. Utilizing a set of instruction models covering all of x86_64 is therefore likely to require considerably longer than that.

However, it is also reasonable to assume that by more carefully selecting the input and output states passed to the SMT solver, thus requiring fewer of them, this runtime can be drastically reduced. Furthermore, the SMT solver is not necessarily required to produce a unique result wherever possible. This task can be delegated to the verification step if the SMT solver can reduce the set of plausible instructions to a manageable size, which can be the case after processing just one input and output state. Whether it is feasible to recover large sections of arbitrary code given this optimization potential is to be answered by further research.

Unfortunately, defending against interrupt-driven code recovery is difficult due to the guest kernel's high privilege level. Defense measures must be implemented in the hypervisor, as any other component of the virtual machine is under the guest kernel's control. A conceivable defense would be to prevent the guest kernel from interrupting the XOM-protected code after just one instruction, rendering it unable to record the output states. If such an interrupt occurs, the hypervisor can transfer control back to the XOM-protected code instead of the kernel's handler routine. After executing one or more additional instructions, the XOM-protected code is interrupted again, at which point the kernel can be invoked securely. However, this defense may endanger the virtual machine's stability, as certain interrupt types require immediate action from the guest kernel. For example, a page fault caused by a memory access cannot be deferred, and continuing to execute code without properly handling it is likely to result in undefined behavior. It also restricts the virtual machine's functionality, as debugging is no longer possible this way.

Another possible defense is Register Clearing. This prevents the kernel from recording a meaningful output state but requires specialized functionality in the XOM-protected code for recovering from interrupts. Consequently, it cannot be applied to most programs, limiting its applicability as a defense. Also, as discussed in Section 5.1.3, Register Clearing imposes a non-negligible performance penalty on the guest, as the hypervisor must be invoked for every interrupt and exception type, even those that the guest kernel could usually handle without the hypervisor's assistance.

Due to the considerable limitations of the available defense measures, the possibility of interrupt-driven code recovery must always be considered when relying on EPT-XOM for security. It can thus be considered as a conceptual weakness rather than an attack vector, making it perhaps the most severe threat to EPT-XOM's confidentiality guarantees. However, it also requires privileges that are typically not granted to user-mode processes and is therefore only a concern in scenarios where the attacker has control over the kernel. Furthermore, Key Locking is not affected by it, as Key Locking implementations always utilize Register Clearing.

5.2.2 DMA Attacks

Direct Memory Access (DMA) attacks leverage hardware with direct access to a system's main memory, thereby bypassing all restrictions set in the page tables [49]. In a typical DMA attack, the attacker uses a malicious peripheral device, such as a rouge Thunderbolt accessory, to obtain secrets from protected memory regions, or to overwrite code for arbitrary code execution on any privilege level. Unfortunately, both MPK and EPT-enforced XOM can be subverted this way. However, a traditional DMA attack requires physical access to hardware, placing it out of scope for many of the protection schemes discussed in this thesis.

Alas, there is a second type of DMA attack that could pose a realistic threat in the context of EPT-enforced XOM if left unaddressed. If a virtual machine has access to a peripheral device, e.g., with PCI passthrough, this device could work as a confused deputy, giving the virtual machine access to memory without properly enforcing page table permissions. While this type of attack typically requires kernel privileges, ways exist to mount such DMA attacks even from user mode.

For example, consider the `CL_MEM_USE_HOST_PTR` flag for OpenCL [50], which affects the behavior of memory buffers for GPU-powered computing. If specified, this flag instructs the GPU to load data directly from a user-supplied pointer, instead of copying it to a driver-controlled location first. As the GPU does not enforce page table permissions when loading memory, an attacker could make a copy of the XOM buffer this way, and then store it to a readable location. Whether this attack works then depends on whether the underlying OpenCL driver correctly verifies the EPT permissions.

Kernel-based attackers, with the capability to communicate with peripherals more freely, can in theory mount similar attacks for every device that can load data through DMA upon request. As long as this data can be read back by the kernel somehow, it is accessible to the attacker.

Fortunately, DMA attacks are relatively easy to prevent on modern hardware. Recognizing the issue of unrestricted memory access for peripherals, chipset and CPU manufacturers

nowadays include so-called IOMMUs in their processors [51, 52]. These hardware facilities apply the concept of virtual memory to DMA requests, thus adding a layer of indirection between DRAM and the peripheral device. Address translation and permission enforcement in the IOMMU are controlled through page tables, which are analogous in concept to the page tables of the CPU's MMU.
It is therefore possible for the hypervisor to ensure that EPT-XOM pages are adequately protected from rogue DMA accesses. DMA reads from EPT-XOM pages can be prevented by modifying the access permission bits, or by unmapping them from the DMA address space altogether. Note however, that while an IOMMU is included in virtually all recent x86_64 platforms, it is often disabled by default for performance reasons. The hypervisor must therefore also ensure that an IOMMU is available in the first place before allocating EPT-XOM pages. With an adequate IOMMU configuration in place though, DMA attacks pose no threat to the confidentiality guarantees of EPT-XOM.

5.2.3 Spectre-like attacks

Spectre is an attack scheme from the family of transient execution attacks, which exploit quirks in a CPU's out-of-order execution system to leak information that would otherwise be inaccessible to an attacker. In particular, Spectre-like attacks aim to mistrain the CPU's branch prediction mechanisms, thus causing it to speculatively execute code or make memory accesses of the attacker's choice. Attackers can then utilize so-called Spectre gadgets in a victim program's code to encode secrets in the cache state, and then leak them through a cache-based side channel attack such as Prime+Probe or Flush+Reload [53]. Many variants of this scheme were proposed in literature, utilizing different mistraining strategies for various branch types [53–56]. Note that this thesis considers any attack following this rough scheme to be Spectre-like, following the classification scheme devised by Canella et al. [55].
A characteristic of Spectre attacks is that they typically cannot bypass memory protection measures such as XOM directly. Instead, they depend on the mistrained branch predictor to cause speculative accesses in a different security domain, where the secret is not protected. A Spectre attack leaking XOM directly is therefore only possible in scenarios where protected pages are mapped as readable in another security domain.
For the EPT-XOM implementation used in this thesis, this can only occur during the *unlock* operation in the hypervisor. During this operation, Xen creates a read/write mapping to overwrite XOM pages before making them available to the guest again. However, the code responsible for clearing the pages can be written in a way that is resistant to Spectre, making it impossible for attackers to encode the data in the cache. The only way to mount a Spectre attack in this scenario is by targeting a separate

hypervisor procedure running in parallel. This is difficult due to the short time window during which leakage is possible, and the inability to repeat the attack once the data is overwritten. In the face of spectre mitigations [34, 57, 58], and further defenses such as randomization of the addresses at which the XOM pages are mapped, the attack becomes entirely unpractical.

Similar conditions apply to MPK-enforced XOM, as a readable mapping for Spectre attacks to access typically does not exist. Furthermore, speculatively changing the `pkru` register in a Spectre attack to retrieve secret information is impossible, as `wrpkru` implicitly linearizes memory accesses, similar to memory fence instructions [9]. A memory access thus cannot occur before the `wrpkru` instruction retires, not even speculatively. Due to this property, MPK was even proposed as a countermeasure against Spectre [59–61], including by Intel themselves, although not in the context of XOM. It is hence impractical at best to leak XOM-protected code directly through a Spectre-like attack, if not impossible.

However, the XOM-protected code can itself be speculatively executed, making it as susceptible to misuse in Spectre attacks as any other code. Consequently, attackers can still exploit instances of careless programming to obtain secrets stored in XOM.

An example of this is shown in Figure 5.7, which shows an example program vulnerable to the *ret2spec* variant of Spectre [54]. The XOM-protected encryption procedure `perform_encryption` stores its key as an immediate value, thus preventing it from unauthorized accesses. With *ret2spec*, an attacker running on the same core can poison the *Return Stack Buffer (RSB)*, which is a specialized cache storing return addresses for optimizing speculative execution. If the encryption procedure is interrupted, only resuming after the attacker process was executed, the last entry in this cache may still contain the return address of the attacker. This causes the CPU to speculatively return to an attacker-controlled address. As the encryption key is still in a register at this point, the attacker can trigger a key-dependent memory access, thereby encoding parts of the key in the cache.

Fortunately, it is possible to mitigate this problem through various means. In this toy program, the easiest fix would be to clear the key from the register state before returning. A more general solution would involve inserting linearizing instructions or retpolines [34] before every indirect branch. Employing branchless programming, where possible, can result in similar protection while circumventing the performance penalty of linearizing instructions. Mounting a Spectre-like attack against a program utilizing these defensive measures is generally not feasible, thus making Spectre a threat only where such measures are not taken.

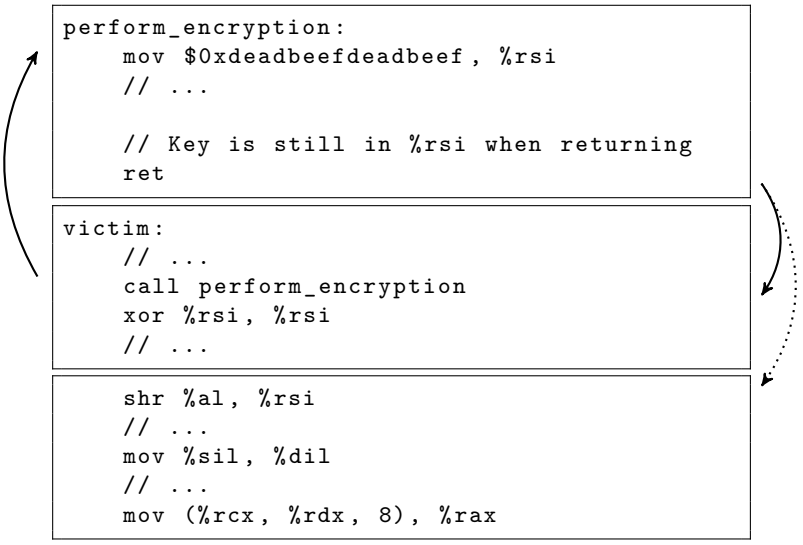

Figure 5.7: An example program vulnerable to *ret2spec*. Solid arrows indicate the architectural execution path, and the dotted arrow shows a speculative path an attacker might be able to induce.

5.2.4 Meltdown-like attacks

In contrast to Spectre, Meltdown-like attacks aim to break hardware-enforced memory protection schemes directly, without relying on code in a different security domain. This family of attacks exploits that on affected processors, page-table settings are lazily evaluated during speculative execution. If a protected value is stored in the cache or another CPU-internal buffer, an attempted access can cause the CPU to speculatively load and use this value, even if the page table permissions forbid this. As with Spectre, attackers can then leak the value through Prime+Probe or Flush+Reload.

The original Meltdown attack as described by Lipp et al. utilizes this to overcome the user/supervisor bit setting, thus allowing an attacker to read arbitrary kernel memory if mapped in a processes address space [62]. Similar attacks make it possible to leak data from a wide range of CPU-internal buffers on affected processors [63–67]. Note that this thesis follows the classification scheme of Canella et al., and thus also considers L1TF attacks and MDS attacks like RIDL to be Meltdown-like [55].

However, leaking data from XOM using a Meltdown-style attack is, for lack of better evidence, improbable to work with any modern processor. While it is possible to bypass MPK protection with Meltdown, this requires the protected value to be cached in either the L1d cache or the store buffer [55]. Such a scenario is unlikely to occur with XOM, as instruction fetches typically do not affect these caches. Other CPU-internal buffers shown to be vulnerable against Meltdown-like attacks, such as the LFB or the load buffer, remain unaffected as well. Instruction fetches only update buffers such as the L1i cache or the higher level L2 and L3 caches, which to the best of my knowledge, are invulnerable

to any known Meltdown-like attack.
I confirmed this assumption with a series of experiments, in which I was unable to apply Meltdown to both EPT and MPK XOM in any capacity, even on affected processors such as the Intel Core i7-7700k. Attempts to leak XOM while cached in the L1i cache, and while executed by another hyperthread on the same physical core, proved unsuccessful. Although this is not definitive proof that XOM is fully unaffected by Meltdown, this is not an unreasonable assumption to make given the lack of contrasting evidence in literature. As such, it may be worthwhile to even consider XOM as a countermeasure against Meltdown. This idea is further explored in Section 6.1.2. Finally, note that modifications made to Meltdown-susceptible buffers by XOM-protected code are still observable to an attacker. They could thereby learn secrets written to memory from within XOM. Note that this is not a concern for Key Locking implementations, as writing secrets to non-XOM memory violates the security model of Key Locking even in the absence of Meltdown-like attacks.

5.2.5 Port Contention

While well-programmed code in XOM cannot be easily leaked through transient execution attacks, it is not exempt from other side channels. However, there are relatively few side channel attacks with the ability to disclose the code itself. Many side channels could leak secrets encoded in XOM, with attackers potentially even observing control-flow and memory-access patterns through cache-based side channels, but the instructions themselves usually remain hidden.
One of the few exceptions to this are *execution unit contention* and *port contention* side channels, which arise from how the CPU processes instructions internally [68]. As part of the decoder pipeline, instructions are first decomposed into semantically simpler μOps, which are distributed to the *execution units* through so-called *execution ports* by the CPU's scheduler. Each execution port is specialized for specific instruction types: On Intel's Skylake architecture for example, four ports per core can handle simple integer arithmetics, but only one can process stores to memory. Although some details vary between manufacturers, this general scheme sees use in virtually all x86_64 CPUs and is well understood due to extensive reverse-engineering efforts [69].
On CPUs supporting *Simultaneous Multithreading (SMT)* (also called *Hyperthreading* on Intel platforms), the execution units in a physical core are shared between two concurrently executing threads. While this leads to more efficient usage of the available hardware resources, and thus to considerable performance benefits, it also has some unintended side effects. If both threads concurrently execute instructions that use the same execution port, one thread has to wait until the other thread's instruction is processed, and the

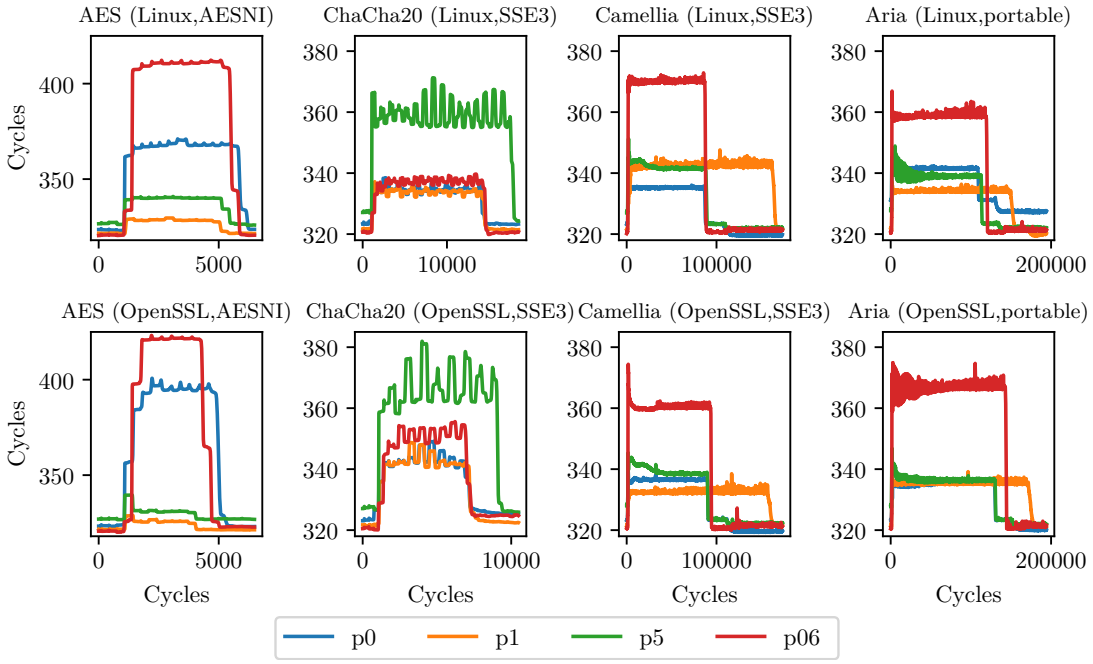

Figure 5.8: Port usage when encrypting 4 kB of data with several cryptographic algorithms on an Intel Core i7-7700k (Kaby Lake) processor. The plots show the mean time needed for executing the port contention primitive at a given time after starting the encryption procedure ($n = 2048$).

port becomes available again. This causes minute, but measurable timing differences, which an attacker can observe to gain information about the execution units that the other thread utilizes. With the *PortSmash* exploit, Aldaya et al. demonstrated that this can be used for recovering encryption keys from another process [68].

To my knowledge, port contention is the only microarchitectural side channel that can reveal information about the code in a processor's execution engine. It is also exceptionally hard to mitigate, as nearly any instruction causes an observable side-effect. However, recovering a victim program's code utilizing this technique is typically infeasible. Firstly, many instructions share the same execution unit utilization profile, and thus cannot be told apart. Secondly, there is no way to recover an instruction's operands. It might be possible, for instance, to determine whether an instruction utilizes a register or memory operand, but the operand itself is unobtainable. Finally, there is little indication to the attacker on where an instruction is located in the victim's code. While an instruction's execution is observable, recovering the location in which this occurs without knowledge of the victim's code is not possible.

Nevertheless, special cases exist in which port-contention side channels can reveal information about the inner workings of XOM-protected code. For example, in cases where one of several algorithms might have been used to solve a specific problem, the developer's choice can sometimes be identified through its port usage profile. As an illustration of this, see Figure 5.8. The contention profiles shown in this figure were created by encrypting

4 kB of data while continuously probing the port usage with another hyperthread. This test was performed on implementations of AES [70], ChaCha20 [71], Camellia [72], and Aria [73] from both the Linux Kernel [37] and OpenSSL [74], all with 128-bit keys. All ciphers were used in counter mode, except for ChaCha20, which is a stream cipher.
Of the 8 execution ports on the Intel Core i7-7700k (Kaby Lake) processor utilized in this experiment, only ports 0, 1, 5, and 6 can be probed reliably, as the remaining ports execute memory operations. The probing primitives consist of instructions that only target a specific port, such as `aesenc` for port 0, `crc32` for port 1, and `vpermd` for port 5. Port 6, which typically handles branches, cannot be probed independently, as every instruction executed through Port 6 also utilizes Port 0. Contention on Port 0 therefore also affects the *p06* curve in Figure 5.8. All probing primitives are designed to take roughly 325 processor cycles if there is no contention. Note that the curves for individual ports are not necessarily synchronized with each other, as probing different ports can slow down the target program in different ways.
From inspecting the profiles in Figure 5.8, it is obvious that the port usage for the Linux and OpenSSL versions of the same cipher are very similar. This makes sense, as both versions implement the same functionality and use the same instruction set extensions. Also, all ciphers except the versions of Aria used in this experiment are implemented in assembly, which prevents compilers from using wildly different instructions due to minor differences in the source code.
Furthermore, the profiles of different ciphers are distinct from each other. This is most obvious with AES and ChaCha20, which most predominantly use ports 0 and 5 respectively. With AES, which is implemented utilizing the AES-NI extensions [9] in both Linux and OpenSSL, this is due to extensive use of the `aesenc` instruction, which is among the few instructions that exclusively utilize port 0 on the Kaby Lake architecture [69]. Both ChaCha20 implementations heavily rely on vector shuffle instructions, which primarily target port 5. The difference between Camellia and Aria is less obvious, as they only significantly differ in the way port 1 is used. Contending this port has a much stronger effect on Camellia, making the encryption take almost twice as long. Aria is far less affected by this. In conclusion, this means that the four algorithms in this experiment can be confidently told apart from each other, even if implemented in slightly different ways.
However, note that this is not conclusive evidence that distinguishing between algorithms based on port usage alone is always possible. Firstly, this specific demonstration only targets Kaby Lake processors, and similar attacks may produce different results on other processor architectures. Secondly, the identifying features of certain algorithms, such as the high usage of port 5 with ChaCha20 in the above example, may also occur with implementations of other algorithms. In such cases, it may still be possible to use a well-trained classifier model to spot more subtle differences, but whether this is practical

remains a question for further research.
Nevertheless, it is not unreasonable to assume that port contention side channels can, in certain cases, aid an attacker in reverse-engineering a protected piece of code. For example, usage of AES-NI produces a highly distinctive port usage profile on Kaby Lake, meaning that an attacker can likely detect whether this instruction set extension is being used. In the face of possible attacks against AES-NI, such as the LazyFP exploit [66], this may inform the attacker about possible attack vectors against the protected program. When designing leakage-resistant schemes around page-locked XOM, potential leakage through port-contention side channels must therefore still be considered.

5.3 Key Locking Case Studies

Whereas Key Locking was already discussed from a theoretical standpoint in Chapter 3, this section aims to demonstrate it is not only theoretically feasible, but also practical. To this end, I present case studies of two concrete Key Locking implementations, one for AES-128-CTR and one for HMAC-SHA256. This involves a discussion of Key Locking's unique implementation challenges and a performance study.
The results show that Key Locking for AES does not reduce performance, with the performance metrics of the Key Locking implementation being comparable to the performance of the well-established *libgcrypt* library [75]. Although the performance of HMAC-SHA256 suffers considerably from the restrictive control-flow protection rules discussed in Section 3.3, the throughput of the HMAC implementation is still large enough to make it practical in many scenarios.

5.3.1 AES-128-CTR

The first Key Locking demonstration implements AES in counter mode (CTR) with 128-bit keys, which is relatively easy to do on modern x86_64 processors due to the AES-NI feature set extensions[30, 70]. With the *aesenc* instruction, processors can perform a round of AES completely in hardware, making it possible to encrypt a 128-bit message block in just ten instructions. Round key derivation is made easy by the *aeskeygenassist* instruction. Because of this, AES on x86_64 requires no complex control flow structures, which makes it easy to implement using only direct branches. Furthermore, both of these instructions can work on AVX register operands, making it easy to derive round keys and perform encryptions without writing key material to memory.
The only real implementation challenge is therefore to handle Register Clearing events gracefully. However, this too is not difficult to do with the CTR mode, as the program can simply check the signal register after encrypting a message block. When the registers

are cleared, it only needs to re-derive the round keys and restore the counter block using the current message offset, which is not confidential and can thus be stored in a register that is unaffected by Vector Register Clearing.
For these reasons, the confidential AES-128-CTR implementation is only 463 bytes long (not including the encryption key), consuming only roughly a 9th of the 4 k code size maximum. Loading a 128-bit encryption key requires 39 bytes of additional code. As the algorithm's code must be stored in the same XOM page as the key, this means that a 4 kB code page can store up to 93 individual AES keys. Also, the small code size makes AES practical to integrate into other algorithms for backing up data to memory, as discussed in Section 3.2.

5.3.2 HMAC-SHA256

The second Key Locking case study implements the HMAC message authentication scheme using the SHA-256 hash function [76, 77]. Although widely supported hardware extensions exist for SHA256 as well, this implementation is far more challenging than implementing AES, as the SHA extensions only cover certain primitive operations [78]. Therefore, the bulk of SHA-256 has to be implemented using more conventional techniques, which involves control flow structures that are far more challenging to create with only direct branches. This implementation should thus provide a more realistic insight into the performance of Key Locking for more involved algorithms.
A special implementation challenge of SHA-256 is that it uses 64 so-called round constants, which are typically read from memory. However, as this may allow an attacker to modify them, potentially undermining SHA-256's security guarantees, they must be stored in code as immediate values, analogously to key. Therefore, each of these constants requires its own small segment of code, which places them in the correct registers. Although these code segments are not large individually, they alone consume about 540 bytes of the 4 kB available and are thus larger than the entire AES implementation.
Another challenge with SHA-256 is that the hash state, which is continuously updated with each message block, is lost after Register Clearing, and cannot be easily recovered. The HMAC implementation must therefore employ encryption to securely store the hash state to memory, as discussed in Section 3.2. In practice, it utilizes AES-NI to save the hash function's progress after every 256 hash blocks, which translates to 16 kB of message data. When interrupted, the hash function can then restore its internal state from the latest checkpoint instead of starting from scratch. This makes it possible to authenticate messages of arbitrary size, even with heavy contention for CPU time from other processes. The encryption key for this is generated at random, and inserted into the program at the same time as the HMAC key. As explained previously, the IV is

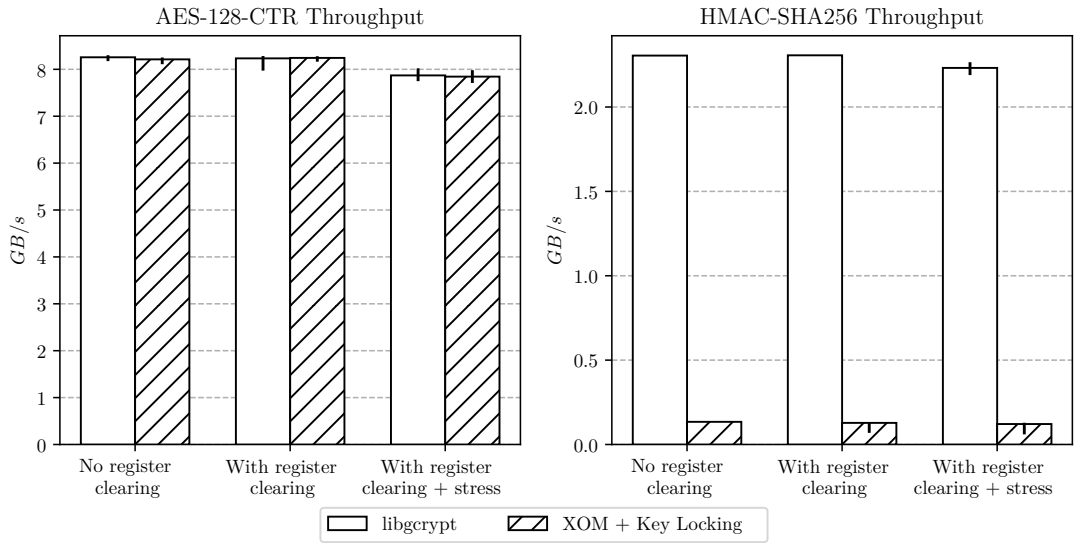

Figure 5.9: Average throughput of the Key Locking implementations compared to libgcrypt ($n = 256$, 512 MB processed per sample, Intel Core i5 13600kf, AVX2/VAES, single thread). The vertical lines indicate the range between the 1st and 99th percentile of samples.

generated with the CPU's hardware random number generator, and saved to regular memory along with the encrypted hash state.

Note that for this specific application, integrity checks are not necessarily required and were thus not implemented. We consider the input to be attacker-controlled, and attackers could thus influence the integrity of the output even if the backup were authenticated with e.g., the GCM block mode.

As a result of these implementation challenges, the HMAC-SHA256 implementation is much larger than the AES implementation, taking up roughly 2.8 kB of XOM. With a 256-bit HMAC key requiring 78 bytes to be loaded, this means that a single 4 kB XOM page can only accommodate up to 16 keys, as compared to the 93 that can fit in a page with AES.

5.3.3 Performance

To assess the practicability of these implementations, I measured their throughput under varying conditions and compared them to the well-optimized implementations in *libgcrypt* [75]. All implementations were tested once with and without register clearing being active in the hypervisor, and once with register clearing and an increased rate of interrupts. Note that only the Key Locking implementations are directly affected by register clearing, as libgcrypt does not use XOM. For the last test, the interrupt frequency is increased by running two separate threads in parallel to the benchmark, one with high CPU usage and one that spins on the *sync* system call. As the test VM

only has virtual cores, this fully utilizes the available CPU resources, while the frequent system calls necessitate a high number of context switches.

The results of this experiment are shown in Figure 5.9. With AES, there is close to no difference at all between libgcrypt and the Key Locking implementation. Even with register clearing and contention for CPU time, the Key Locking implementation is only 0.33 % slower on average.

This makes sense, given that the instruction sequence for encrypting data with AES-NI does not allow for much variation, and is thus nearly identical between the two implementations. Since the program does not cross code page boundaries, overhead due to EPT-XOM address translation does not impact the performance. As mentioned previously, recovery in the case of register clearing is very easy, as encrypted blocks are written to memory immediately, and counter mode allows for restarting the encryption at any offset with little performance loss. The most important result here is that these recovery procedures do not significantly impact the overall performance, even under heavy contention.

Unfortunately, the results for HMAC are less impressive, with the Key Locking implementation's throughput being more than an order of magnitude lower than libgcrypt's. This is not unexpected, as the SHA256 code saw drastic alterations to follow the control-flow protection rules from Section 3.3. For example, one of the largest changes was to encode the round constants into code, which significantly increases the amount of code executed per round and makes less efficient use of the CPU's caching structures. Another change was the introduction of the state-backup system, which adds additional overhead.

Also, the effect of register clearing on the HMAC implementation is much greater than with AES, as the recovery procedures are significantly more costly. Even without added contention for CPU time, the HMAC implementation's throughput is 4.6 % lower on average when register clearing is enabled. For comparison, AES is even 0.2 % faster when register clearing is enabled, indicating that within measuring precision, there is no performance difference at all. Additionally, whereas the added CPU contention slows down libgcrypt's implementation by only 3.2 %, the throughput of the Key Locking implementation is reduced by 5.3 %. This makes sense, as up to 16 kB of progress can be lost upon an interrupt.

Nevertheless, HMAC with Key Locking is not entirely unpractical. While the significant overhead may be unacceptable in performance-critical applications, the HMAC implementation is still capable of processing roughly 121 MB per second, even with frequent interrupts. In scenarios where hiding the key is more important than absolute performance, HMAC with Key Locking is therefore still a viable option. It should also be noted that the HMAC implementation still leaves much room for optimization, as it was designed with security in mind rather than performance.

In summary, the most important results of this experiment are as follows:

- Key Locking does not impact the performance of AES-NI by a significant margin, even with frequent interrupts.
- The performance of more involved code can be drastically reduced, as demonstrated with HMAC. Nevertheless, this does not render Key Locking implementations entirely unpractical.
- Although programs with Key Locking can be more affected by a high interrupt frequency than other programs, this effect is not so drastic as to make them unusable.

Chapter 6

Discussion

6.1 Potential Applications

This section lists and discusses potential applications for XOM, and XOM-based Key Locking in the context of x86_64. The focus here lies more on techniques that are entirely novel or have yet to be applied to x86-based platforms. Therefore, XOM-based defenses against code-reuse attacks are not listed here (see Section 7.2 instead).
In summary, XOM is immensely useful as a mitigation against memory disclosure attacks. This can be applied to defense measures against Spectre and Meltdown attacks, but also strengthen the security of conventional cryptographic software and Digital Rights Management schemes. In certain scenarios, the security guarantees of EPT can also provide tamper resistance, and help defend against reverse-engineering attempts.

6.1.1 Leakage Resistance for Encryption Keys

EPT-XOM-based Key Locking provides strong leakage resistance for encryption keys. Keys can only be leaked during the short time window in which they are initialized. If an exploit requires a large runtime, as is the case with many transient execution attacks, or in cases where an attacker only gains control after a key is in place, leakage is impossible. The attack surface can be even further reduced by initializing encryption keys in the hypervisor, eliminating any way to leak them from within the guest. As Key Locking for AES incurs little to no overhead, this may be used as a general hardening method against key leakage. Nearly any software that uses AES, which includes most modern applications with networking capabilities, may benefit from this.
Another advantage of this is that the permission to use the encryption key can be revoked at any time, which is obviously no longer possible once the key has leaked. This property

might prove useful in a variety of applications. For example, consider an encrypted hard drive, which contains sensitive data that an attacker might want to steal. In the case that monitoring software detects an ongoing attack, it can simply unmap the XOM page containing the encryption key. This might still allow an attacker to partially leak the hard drive's contents while the XOM pages are present, but once the encryption keys are gone, the attacker has no way to access the remaining files on the hard drive, even if they fully compromised the guest.
Another advantage of EPT-enforced Key Locking is that it provides strong protection against side channel and transient execution attacks. While these properties are not strictly limited to encryption keys (see Section 6.1.2), encryption keys are a valuable attack target and thus greatly benefit from this leakage resistance.

6.1.2 XOM as a Defense against Microarchitectural Attacks

Hardening large software projects against transient execution attacks such as Spectre is difficult. Existing countermeasures like Intel's Indirect Branch Restricted Speculation (IBRS) can severely degrade performance, effectively crippling certain workloads [46]. More lightweight mitigations like Retpoline, on the other hand, were shown to be inadequate against certain Spectre variants [56], while still inducing considerable overhead [79]. Many Meltdown-style attacks, such as those from the class of Mircoarchitectural Data Sampling attacks, cannot be easily mitigated at all without severely restricting an affected processor's functionality.
However, as discussed in Section 5.2, XOM is surprisingly resilient against transient execution attacks. Meltdown-style attacks require data to be stored in specific CPU-internal buffers, none of which are directly affected by instruction fetches. Spectre-style attacks require secrets to be readable in a different security domain, which is not the case with XOM. Schemes utilizing these properties may therefore provide a strong defense against transient execution attacks at little to no cost. Note that the stronger security guarantees of EPT-based XOM are not necessarily required here, as MPK-enforced XOM cannot be speculatively read either.
The easiest method to leverage XOM for storing secrets is to encode them into intermediate values for *mov* instructions. For example, a secret encryption key can enter the register state this way without causing any microarchitectural side-effects that might reveal its value to an outside observer. Applications for this were already discussed in Section 6.1.1. However, this property is not restricted to encryption keys and can be applied to many other scenarios. With certain secrets, it may not even be necessary to load them into the register state at all, which further reduces the attack surface for Spectre attacks.

```
1  is_hash_correct:
2      mov $1, %rax
3      xor %r8, %r8
4
5      cmp $0x12345678, %edi
6      cmovne %r8, %rax
7      shr $32, %rdi
8      cmp $0x90abcdef, %edi
9      cmovne %r8, %rax
10
11     cmp $0xcafebabe, %esi
12     cmovne %r8, %rax
13     shr $32, %rsi
14     cmp $0xdeadbeef, %esi
15     cmovne %r8, %rax
16
17     // Repeat for the remaining hash
18     // ...
19
20     ret
```

Figure 6.1: Checking whether a password hash is correct without loading the correct hash into non-XOM memory or the register state (AT&T syntax). The 256-bit hash to be checked is passed as 4 seperate 64-bit values, which are stored in (`%rdi`, `%rsi`, `%rdx`, `%rcx`).

For example, consider the hash of the root user's password, which is a popular target for Spectre-like attacks on Linux. This hash is only used in one way: When the user makes an authentication attempt, the authenticator checks whether the hash of the password the user entered matches the correct password hash. Apart from this, the correct hash is not used in any kind of computation or algorithm.

Instead of encoding the hash into *mov* instructions, it is therefore possible to encode it as a series of *cmp* instructions. The resulting code can then be executed to check whether the user's hash matches the correct hash. See Figure 6.1 for an example of how this could be implemented for a 256-bit hash value.

If done correctly, this reduces the attack surface to a minimum, as the hash is only in readable memory or the register state when the function is initialized, and when an authentication attempt with the correct password is in progress. The short time window in which this occurs makes most transient execution attacks impractical, especially if the code is also otherwise hardened against maliciously induced speculation.

Nevertheless, this approach is not without its limitations. In contrast to IBRS or Retpoline, which can protect an entire program, XOM-based schemes only protect individual values. While effective for encryption keys or password hashes, applying this concept to entire memory regions is impractical, if not impossible. Certain types of attacks, such as those breaking ASLR by disclosing code pointers with Spectre, are therefore still possible.

6.1.3 Copyright Enforcement and DRM

Digital Rights Management (DRM) systems aim to enforce the copyright associated with remotely distributed media, typically by preventing unauthorized copying through cryptographic means. This leads to a highly unusual threat model, in which devices must protect media files from their rightful owner, instead of an attacker in the traditional sense. To achieve this, DRM systems usually encrypt media before it is delivered, and ensure that decryption can only occur in a highly controlled environment to prevent misuse of the encryption key.
Where supported, they utilize the isolation and remote attestation features of TEEs for this purpose [80]. On mobile devices, where TEEs like ARM's TrustZone are widely supported, this scheme sees widespread use. However, in environments where TEEs are unavailable, DRM systems instead have to rely on software-only mechanisms, which can at best obfuscate how keys are managed. Software-only DRM systems are thus prone to a variety of attacks, many of which were even demonstrated in practice [80].
While XOM-based techniques like Key Locking cannot solve this issue in its entirety, they can mitigate many problems that DRM systems typically face in the absence of a TEE. Most importantly, an encryption key that is locked into EPT-XOM cannot be retrieved again, not even by the kernel. Leaking the encryption key is thus only possible by tampering with the hypervisor. This is far more difficult than manipulating the kernel of a rich operating system, as hypervisors like Xen typically do not allow users to install driver modules, and lack debugging or introspection features. Hence, guests cannot alter a running hypervisor's behavior without exploiting security vulnerabilities, meaning that keys managed by the hypervisor are fully inaccessible.
DRM systems can utilize this property to protect encryption keys reliably. In such a scheme, the client-side of media key exchanges occurs within the hypervisor, ensuring that the keys are never directly accessible from within a guest. These keys are made available to guests as XOM-protected code segments, thus allowing them to use the keys while preventing unauthorized disclosure. However, to guarantee the security of this scheme, it is also necessary to solve the following challenges:

- The integrity of the hypervisor upon startup must be guaranteed.
- There must be a way to reliably establish a shared root of trust between the media distributor and (only) the hypervisor.
- Measures must be in place to prevent malicious usage of the XOM-protected code segments.

The first challenge is relatively easy to solve if a TPM is available. In this case, it is possible to remotely attest the software stack's integrity to the media distributor, allowing

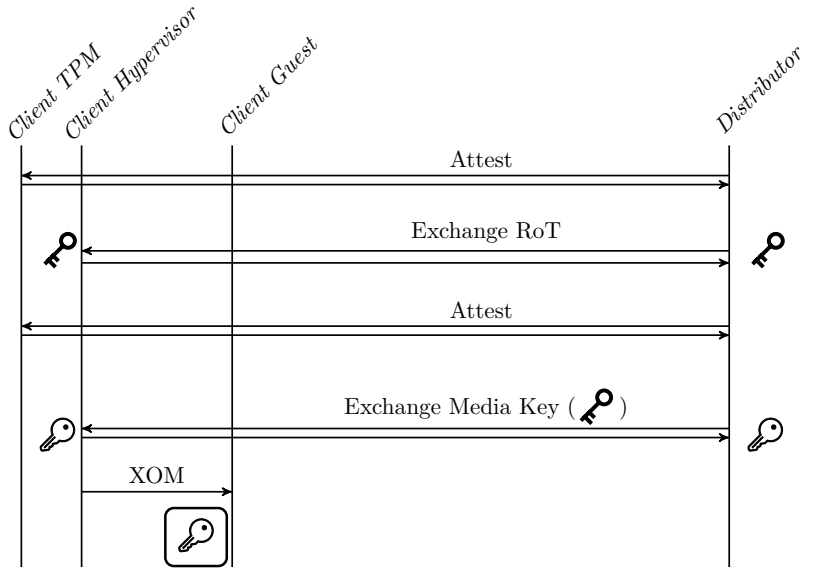

Figure 6.2: A high-level overview of the proposed key exchange for DRM. The media distributor and the client hypervisor first exchange a shared Root of Trust ( ), which is later used to exchange media keys ( ). These media keys are made available to guests with Key Locking.

them to decide whether they trust the user's configuration. In the absence of a TPM however, this cannot be reliably verified, which constitutes a major limitation of this scheme.

The second challenge, although less straightforward, can also be solved with a TPM. Distributors can verify the integrity of the software stack down to the DRM system's user mode process, guaranteeing that it relays the key exchange messages to the hypervisor instead of handling them in the kernel or in user mode. This must occur before and after the root of trust is exchanged, to guarantee the system's integrity during the entirety of the process. See Figure 6.2 for a high-level schematic of how key-exchanges work in such a system.

After this root of trust is in place, no further attestation is necessary. Since the hypervisor's behavior cannot be altered except through a vulnerability, the distributor can confidently assume that any keys exchanged based on the shared root of trust are handled in the intended manner. As mentioned before, this would not be the case with the kernel of a feature-rich operating system, as their behavior can be easily altered after the root of trust is exchanged.

This leaves only the third, and perhaps most difficult challenge: Ensuring that the guest instances do not use XOM-protected keys to decrypt files in unintended ways. The easiest way to solve this problem would be to forego the use of XOM entirely, integrating the entire decryption logic into the hypervisor. However, this is usually not an option, as hypervisors strive to be as minimal as possible. Integrating an entire DRM system directly into the hypervisor would massively contribute to bloat, and affect all users by

increasing the hypervisor's memory footprint. It is also not exactly generic, as different proprietary DRM systems would require different hypervisor implementations. A simple key exchange protocol, on the other hand, is generic and requires relatively little code to implement, making it the preferable option.

The only way to prevent misuse of the XOM-protected code is therefore either to rely on the kernel and user-mode components of the DRM system or to integrate context validation checks into the XOM-protected code itself. The first option, while difficult to circumvent if done right, is not entirely reliable, as the guest's state can theoretically be altered by the user in arbitrary ways through kernel drivers and debugging features. A sufficiently motivated attacker might therefore find a way to disable these mechanisms before they can unmap the XOM pages, allowing them to decrypt media without authorization. The second option, although harder to manipulate, is more restricted in its capabilities. While it might for example be possible to embed a TSC-relative timestamp into the XOM page, making decryption only possible at certain times, more involved context validation is made difficult by the various rules the XOM-protected code has to follow to prevent control flow manipulation. Therefore, neither of the approaches is perfectly reliable, although they can make misuse more difficult.

However, even if the XOM-protected code were perfectly protected from misuse, this hypervisor-based DRM scheme would not be without limitations. Firstly, it depends on the presence of a TPM for integrity verification, which is not always given. It therefore only provides tangible security guarantees in a setting where a TEE is not available, but a TPM is. Even if a TPM is present, reliably attesting the system's software stack is difficult, and requires extensive software support. Secondly, the scheme assumes that the behavior of a running hypervisor is hard to manipulate. While this is true for thin hypervisors, or standalone hypervisors like Xen, it does not hold for hypervisors that are part of a larger operating system, such as the Linux KVM or Microsoft's HyperV.

Another issue is that the decrypted media ends up in readable memory at some point, as playing it correctly would hardly be possible otherwise. While TEE-based DRM schemes are prone to this as well, they do not have to adhere to the highly restrictive rules XOM-protected code has to follow to ensure confidentiality, and can therefore execute much more involved programs in a trusted environment. This allows them not only to decrypt media files securely, but also to decode them into analog data such as frame buffers and audio signals before making them available to less trusted code. Although it can be captured with external hardware or special software tools, re-encoding this analog data into a redistributable format induces a loss of quality, which does not occur when XOM-protected code is used for decrypting the files only.

However, despite the many weaknesses and pitfalls that come with a XOM-based DRM scheme, XOM can still be immensely useful as a hardening feature. Although misuse of the XOM code is difficult to prevent entirely, it significantly raises the difficulty level

of decrypting a DRM-protected file compared to software-only solutions. Even if the hypervisor is not a trusted component or the key exchange occurs in the guest, actually leaking decryption keys requires significantly more effort than simply reading it from kernel-accessible memory. Note that even the most involved DRM schemes cannot fully prevent unauthorized redistribution, as it is always possible to capture and reencode copyright-protected media with external hardware. Therefore, DRM systems should be seen more as a deterrent than a reliable security measure. In this context, any technique increasing the difficulty of unauthorized redistribution at little cost might prove useful.

6.1.4 Tamper-resistant Code

A noteworthy side effect of EPT-enforced XOM is that it provides near-absolute tamper resistance. Once locked, no modifications to an EPT-XOM page are possible, apart from releasing it again. A similar mechanism can also be created with EPT-enforced read-only pages, if used similarly to Page Locking. Potentially, this may prove useful in scenarios where an attacker abuses interfaces such as the *WriteProcessMemory* function on Windows [81] or *ptrace* on Linux [37] to manipulate a program's runtime behavior.
A typical example of this is cheating in online video games. Some players may seek an unfair advantage by modifying the game's memory state, for example in making the opponent's character model visible through walls in a competitive shooter-style game. In an effort to prevent this, many games employ self-monitoring code to detect such modifications. Over the years, this has led to an arms race with cheat software developers, who utilize increasingly sophisticated methods to disable any self-monitoring logic through further memory manipulation. In turn, anti-cheat software now often requires kernel privileges to supervise the game's state externally, which has raised numerous concerns about privacy and security in the past [82].
EPT-enforced page locking can eliminate the need for privileged supervision in this context. If the game's code is placed in XOM, attempted alterations are prevented by hardware, even if performed by the kernel. Therefore, self-monitoring software can run as part of the game's process without being at risk of external intervention. Modifying the code is not possible, and modifying the non-XOM memory segments triggers the now unalterable self-monitoring logic. This makes any external modification of the game's memory state challenging at best, given that the self-monitoring code is well-programmed and cannot easily be subverted by data-only attacks.
While there is no guarantee of the code's integrity upon startup, validating the code against a cryptographic signature in the operating system can provide these guarantees. These integrity checks can be implemented generically without knowledge of the game's internal workings, and many operating systems already implement similar integrity checks

for other security-related purposes [83]. Through the use of a TPM, it is even possible to remotely attest the integrity of the operating system itself, making it possible to deny service to players with an untrusted software stack. A kernel module provided by the game developer is therefore no longer required.
When applied to other types of software that might benefit from tamper-resistance, such as antivirus programs or user-mode authentication mechanisms, this scheme unfortunately proves less useful. In contrast to video game cheating, the attacker's goal in this scenario is usually not to modify the program's behavior in a specific way, but to disable or circumvent it altogether. As access to the interfaces described above is typically restricted to privileged users and the user of the target application, an attacker with access to them has many other means at their disposal to achieve this effect. Antivirus software can simply be stopped, and privilege escalation through subversion of an authenticator makes no sense if the authenticator and the attacker already share the same privilege level. The effectiveness of EPT-enforced memory protection in this scenario is therefore also limited.

6.1.5 XOM as a Defense against Reverse Engineering

Another prevalent use of XOM is as a defense measure against reverse engineering. This is particularly common in ARM-based embedded systems, where manufacturers sometimes make an effort to hide the inner workings of their firmware [1]. This idea is also applicable to x86_64. For example, an Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) provider might be interested in letting their clients use a proprietary shared library while protecting it against reverse engineering. EPT can provide this protection if the hypervisor only makes these libraries available to instances in the form of XOM. However, as clients are typically given kernel privileges, or at least debugging capabilities in their IaaS instance, XOM cannot provide strong security guarantees in this setting. Interrupt-driven Code Recovery attacks are capable of restoring a protected program with a high degree of accuracy. The only reliable countermeasure is Register Clearing, which imposes a performance penalty on the entire guest system and requires special recovery measures in the library to prevent crashes. There is also the issue of data written to readable memory by the library, which can aid an attacker in their reverse-engineering effort. For example, by monitoring the stack, they could access return addresses and thus recover the call trace. Finally, cache-based side channel attacks can reveal information about the library's control flow structure and data access patterns.
Nevertheless, XOM might be of use as a hardening technique. While it cannot prevent reverse engineering entirely, any reverse engineering effort becomes considerably more difficult if the code is XOM-protected. For example, although Interrupt-driven Code

Recovery can accurately recover a program, the vast computational resources required for mounting such an attack on large stretches of arbitrary code could make it non-viable for many attackers. In combination with other obfuscation techniques, XOM can therefore serve as a strong deterrent against reverse engineering. The only major downside of XOM in this context is the slight performance overhead of address translations. However, depending on how critical the performance of the protected library is to the hosting provider, this might not be an issue.

6.2 Limitations

6.2.1 Limitations of EPT-enforced XOM

While EPT provides strong security guarantees, it is not free of limitations. Perhaps the most obvious drawback of EPT is that it is limited to virtual machines, which prevents its usage in fully native environments. However, this limitation is easy to overcome with a so-called thin hypervisor, which only virtualizes a single guest. Using hardware-assisted virtualization, thin hypervisors can be implemented with as little as 2.5 KLOC, and thus have a very small attack surface [84]. Also, some operating systems, such as Windows, already incorporate virtualization into their security model [85], making the adoption of EPT-based XOM somewhat easier.

Another limitation is that EPT is only available on Intel platforms. While AMD's SEV-SNP extensions make SLAT-enforced XOM possible as well, these extensions are less widely supported than EPT. Therefore, Intel users are more likely to benefit from the security guarantees of SLAT in this context.

Moreover, the security of EPT depends entirely on the integrity of the hypervisor. If an attacker can somehow manipulate the hypervisor's behavior, for example through a memory safety issue, they can easily disclose secrets protected by EPT. Schemes like Key Locking, which depend on EPT-XOM for the confidentiality of data, are therefore inherently weaker than schemes utilizing hardware-based features like TEEs or Intel's Key Locker. Where available, these features should therefore be preferred over XOM-based Key Locking. However, note that hypervisors are typically much harder to compromise than the kernel of a feature-rich operating system, and can thus still provide a high level of security.

Finally, there is the performance overhead of code address translations with EPT-enforced XOM. As shown in Section 5.1, this can have a noticeable impact on real-world software, making EPT-XOM somewhat ill-suited for protecting entire programs in performance-critical applications. This effect can likely be mitigated with 2 MB code pages, although this was not experimentally verified. Applications like Key Locking, where the code

cannot extend beyond code page boundaries for security reasons, are unaffected by this as well.

6.2.2 Limitations of Key Locking

Besides the limitations of EPT-XOM, Key Locking in particular has some additional limitations that are noteworthy. Firstly, Key Locking is only secure if programs follow the highly restrictive rules from Section 3.3. While this is easy with AES-NI, the HMAC implementation shows that the performance of more involved code can suffer from these restrictions. It should also be noted that programming code that is secure in the context of Key Locking is impossible with any conventional programming language. To reliably ensure that the control flow protection rules are upheld, programmers must use assembly languages instead, which makes the development of more involved Key Locking implementations challenging and error-prone.
Furthermore, Key Locking depends on the L1 iTLB not being shared between hyperthreads. As discussed in Section 3.3, the guest kernel could otherwise map a different code page to the EPT-XOM page's virtual address while it is being executed, thus hijacking control flow. This property is not always made public by CPU manufacturers, and must instead be reverse-engineered by security researchers. Therefore, it is sometimes unclear whether it is safe to depend on EPT-XOM for confidentiality. On CPUs where the L1 iTLB is known to be shared, users should not depend on Key Locking for dependable security at all, although it might still be useful as a hardening feature.

6.3 Potential for Future Work

6.3.1 XOM as a Spectre Mitigation

Although this thesis discusses XOM as a Spectre mitigation and provides a foundation for building such defenses with *libxom*, no effort is made to integrate this defense into real-world software. As modifying the cryptography code of projects like OpenSSL or the Linux kernel on such a fundamental level is a highly time-consuming process, and this work's focus lies more on evaluating attack vectors and conceiving novel applications for XOM, this is not further pursued. However, studying the implementation challenges and performance impact of this defense may provide an interesting research opportunity, since the practicability of XOM as a Spectre defense largely depends on these aspects.
Furthermore, it may be interesting to perform a more systematic study on which types of secrets can be protected this way. This thesis covers encryption keys and data used for authentication, like password hashes. However, XOM could also be useful in

protecting other kinds of data, for example in the environment of a web browser. This includes Session IDs, passwords, API tokens, and sensitive personal data, to name a few. As web browsers are popular targets for Spectre attacks, investigating the feasibility and practicability of XOM-based defenses in this context may provide further research opportunities.
Finally, XOM as a Spectre defense may be applicable to architectures other than x86_64. The only precondition is that data loads from XOM are not transiently executed. Whether this is the case for other architectures that support XOM, like AArch64, is yet to be determined. Note however that popular adoption of XOM on AArch64 currently seems unlikely, due to its role in breaking Privileged Access Never (PAN), ARM's equivalent to Supervisor Mode Access Prevention [86].

6.3.2 Key Locking

As with XOM as a Spectre defense, no efforts are made to integrate Key Locking into real-world software, therefore providing further research opportunities. Due to the application potential of Key Locking in the context of DRM systems, it may also be interesting to investigate the implementation challenges of a scheme as described in Section 6.1.3.
However, perhaps the most intriguing research opportunity lies in the investigation of Key Locking for public key cryptography. Implementations for public key signature schemes that do not write key material to memory already exist (see Section 7.4), although they do not follow the strict control flow protection rules for Key Locking implementations. The main question in this context is whether such an implementation is practical. HMAC with SHA256, which is arguably easier to implement, and less affected by the programming restrictions, already suffers a dramatic performance penalty with Key Locking. With schemes like RSA or Ed25519, it is therefore possible that a Key Locking implementation is so slow that it becomes entirely unpractical. If deemed practical, however, Key Locking could be useful in a variety of additional protection schemes, and provide leakage resistance to private keys.

Chapter 7

Related Work

7.1 Other Ways to Create Execute-Only Memory on x86_64

While the aforementioned approaches using MPK and SLAT are, to the best of my knowledge, the only ways to enforce execute-only memory purely in x86_64 hardware, there are other methods to create it with the assistance of software-implemented fault handlers.

Sparks and Butler propose *ShadowWalker* [87], a technique for hiding kernel rootkits using memory subversion. ShadowWalker creates a so-called *split TLB*, in which the state of the instruction TLB differs from the state of the data TLB. This is achieved through a special page fault handler, which yields different address translations depending on whether the access type is read or execute. Once set up, the split TLB causes read accesses to be translated differently from execute accesses, allowing a kernel rootkit to hide its code from detectors. The concept of a split TLB also allows for creating execute-only memory, with read accesses yielding static data instead of code [4]. However, despite a design with separate data and instruction TLBs being the de-facto standard for x86_64, many recent CPUs also feature a shared last-level TLB, which caches translations for both execute and read accesses [30]. This makes the split TLB approach unreliable on recent hardware [88], hence why it is not considered in this thesis.

Backes et al. propose a different technique [2], which keeps a *sliding window* of the most recently used code pages readable, while the rest is marked as non-present. Once a new, non-present code page is accessed, the page fault handler determines whether a read or execute access occurred, and maps the page only in the latter case. The last code page in the sliding window is then marked as non-present again, to keep the window's length constant. While this mechanism is suitable as a defense against code-disclosure

attacks aiming to disclose large portions of code, for example as a prerequisite for return-oriented programming, small portions of code remain readable at all times. It is therefore unsuitable for applications like Key Locking, where keeping even small segments of code readable would undermine security.

7.2 Execute-Only Memory as a Countermeasure for Code-Reuse Attacks

Perhaps the most obvious application for execute-only memory is the prevention of code disclosure, which is often necessary as a precursor for code-reuse attacks. Exploitation techniques like Blind-ROP [18] and JIT-ROP [17] all involve a code-disclosure step. XOM was therefore proposed as a countermeasure against these attacks, by Backes et al. [2], with schemes like HideM [4], NEAR [89], and Heisenbyte [90] reiterating the idea with different XOM-enforcement techniques.

However, simply protecting code with XOM is often an inadequate defense against code-reuse attacks, as the code layout of publicly available software is usually known to an attacker. JIT-compiled code is also affected by this, as attackers can leverage code that is compiled in a predictable way [91]. This issue led to the emergence of *leakage-resistant code diversity* schemes, in which code undergoes a *diversification* process before being loaded into XOM. These schemes randomize the program's layout on a fine-grained level, necessitating either code or code pointer disclosure to successfully mount a code-reuse attack.

One of the most comprehensive techniques in this family is *Readactor*, which was proposed by Crane et al. in 2015. Readactor combines code diversification and XOM with code-pointer hiding techniques, preventing an attacker from disclosing the code layout through pointer leakage. Diversification with Readactor involves a variety of techniques, such as function permutation, equivalent instruction substitution, stack layout randomization, and randomized insertion of `nop` instructions. This ensures that every instance of a Readactor program is unique so that a working gadget chain for one instance cannot be used on another. The layout of the code is then hidden using EPT-based execute-only memory. Additionally, to prevent an attacker from making inferences about the code's layout by leaking code pointers, all code pointers stored in memory point to a XOM-protected trampoline, thus hiding the target code's true location. In combination, these primitives make return-oriented programming near-impossible, as an attacker is unable to retrieve any meaningful information about the code's layout, even with a powerful disclosure primitive.

However, Readactor's extensive diversification measures also impose a non-negligible performance overhead on the program, with an average runtime increase of 6.4 % for the

SPEC CPU2006 benchmark. Furthermore, while code pointers themselves are protected from unauthorized disclosure, trampoline pointers are still stored in readable memory. The assumption here is that they cannot easily be associated with the function their trampoline points to. Rudd et al. proved this assumption wrong by demonstrating that a trampoline pointer's location in memory is often enough to determine its target function, thus allowing for whole-function reuse attacks [92]. While subsequent work like R^2C addresses this by also randomizing the data layout, this comes at the cost of an even greater performance overhead [8].

Several proposals were made to apply Readactor-like diversification schemes to kernel code [5–7]. As there are many ways in which attackers could leak code from the kernel, all of these proposals leverage XOM in some form as a means to prevent this. However, none of these schemes have seen popular adoption in real-world applications to date, likely due to the inherent impracticability of diversifying the kernel for every system individually.

The main contribution that this thesis makes in the context of these schemes lies in providing performance metrics for the different XOM-enforcement methods. XOM enforced through EPT can cause runtime overhead on a microarchitectural level. Therefore, MPK should be the preferred XOM-enforcement mechanism where available, despite its weaker security guarantees. In combination with a well-designed diversity scheme, this should not make much of a difference, as an attacker must first locate a *wrpkru* gadget to break MPK. If this is not possible, the security of MPK should be just as strong as the security of EPT. However, as these guarantees largely depend on the design of the underlying diversity scheme, a research topic worthy of its own thesis, this work does not further investigate these properties.

7.3 XOM for Protecting Intellectual Property

Many modern microcontrollers with integrated flash memory features some sort of readout protection mechanism. Firmware in particular is information that hardware manufacturers often seek to protect, as the development of this code is costly, and may give competitors a commercial advantage if they can obtain it. However, this creates a conundrum: How can software be executed while preventing unauthorized readouts at the same time? Many hardware manufacturers choose to employ XOM as a means to solve this problem. While not particularly common on x86_64, XOM is therefore widely used and deployed in ARM-based embedded systems [1].

This widespread deployment sparked research efforts into the security of XOM-based defenses. In 2019, Schink and Obermaier proposed Interrupt-based Code Recovery, which is perhaps the most comprehensive attack on XOM to date [1]. This thesis already

discusses Interrupt-based Code Recovery in depth, and devises a modified version suitable for complex instruction sets like x86_64 (see Section 5.2.1). The primary difference between this modified version and the original approach by Schink and Obermeier is the use of a constraint solver. Whereas the original version simply enumerates plausible instruction instances based on a set of primitive rules, the complex nature of x86_64 demands a more sophisticated means of deriving instructions. It is therefore necessary to incorporate detailed knowledge about instruction semantics into this process, which a constraint solver can do with relative efficiency. While this is computationally expensive, the results of this thesis demonstrate that Interrupt-driven Code recovery is not only a threat on RISC architectures but also on CISC architectures like x86_64.

7.4 Memory-less Encryption

While this work utilizes encryption without writing secrets to readable memory, it is by no means the first to do so. Initial efforts to design such schemes were made to defend against cold-boot attacks, which utilize the property that DRAM retains its contents for a short time after losing power. This enables attackers with physical access to a device to restart it, and then dump a memory image to obtain cryptographic secrets that would otherwise be protected by software. Therefore, a scheme that does not write cryptographic secrets to DRAM mitigates such attacks. Note that XOM-based Key Locking is not a mitigation against cold-boot attacks, as the key is ultimately still stored in DRAM.

Müller et al. propose TRESOR, which stores encryption keys in the CPU's debug registers [93]. These registers are inaccessible without kernel privileges and reliably reset their contents upon a reboot. As with this work's AES implementation, encryption with TRESOR occurs solely in the CPU's register state, utilizing AES-NI. However, while TRESOR successfully protects against cold-boot attacks and unprivileged local attackers, privileged attackers can still access the debug registers and thus disclose the key. TRESOR is hence not a scheme suitable for replacing TEEs.

Works like PRIME [94], Copker [95], and Mimosa [96] apply the concept of memory-less encryption to public key cryptography. All of them utilize TRESOR to encrypt key material before storing it to memory, similar to how this work's HMAC implementation encrypts its internal state. During the encryption process, PRIME uses the AVX registers to store the intermediary results, whereas Copker uses the L1D cache. However, this limits PRIME to small key sizes, while Copker's security strongly depends on poorly documented behavior of the underlying hardware. Mimosa utilizes hardware transactional memory instead, which is typically implemented in the CPU and thus not written to DRAM. However, hardware support for this feature is rare, as it was disabled on

many Intel processors to mitigate TSX asynchronous abort attacks [64]. Although still supported by many recent data center CPUs, consumer-focused processors generally lack this feature altogether.

Memory-less implementations for elliptic curve cryptography schemes also exist. Yang et al. demonstrate that the ECDH key exchange algorithm can be implemented within the CPU's register state without performance overhead [97]. As with TRESOR, private keys in this scheme are stored in the debug registers. Fu et al. propose RegKey, which partially implements memory-less elliptic curve signature algorithms [98]. RegKey only performs simple prime field operations in the registers, whereas the more involved elliptic curve group operations still utilize regular memory. This is sufficient to ensure that private keys are not stored in DRAM as plain text. However, RegKey only protects against one-shot memory disclosure attacks, with persistent software-based attacks still allowing for key disclosure.

Chapter 8

Conclusion

At its core, XOM is a simple and lightweight protection scheme. This simplicity makes XOM highly versatile, and easy to implement in hardware. As such, it is surprising to see how rarely it is used in practice.

The main conclusion of this thesis is that the application potential of XOM on x86_64 is much broader than previously assumed. For example, XOM is useful not only in mitigating code reuse attacks but also in mitigating Spectre and Meltdown attacks. Furthermore, Key Locking can provide security guarantees that would usually require a TEE, achieving strong leakage resistance for encryption keys on hardware where such facilities are unavailable. Although not as strong of a defense as TEEs, this may be of use as a hardening feature in DRM systems. Finally, EPT-based XOM can help against unauthorized reverse engineering and even provides tamper resistance in certain scenarios.

The only factor that significantly limits the usefulness of these schemes is poor or missing hardware support. While workarounds using EPT or MPK can make XOM available to software, these solutions are far less practical than protection schemes with proper MMU support, like $W \oplus X$. Nevertheless, with MPK support in particular becoming more common in recent years, it is entirely possible that XOM will see more widespread adoption in the not so distant future. Additionally, techniques like Page Locking leverage not only EPT but also other hypervisor-specific features like Register Clearing. Therefore, the lack of secure XOM enforcement methods in non-virtualized environments is not an issue in for these applications.

Despite all limitations, XOM's utility as a security hardening feature therefore remains substantial. While subversion of the discussed schemes is often possible in theory (e.g., by manipulating the hypervisor or through attacks like Interrupt-driven Code Recovery), they make practical exploitation significantly more challenging. Given XOM's low runtime overhead, it is therefore hard to deny its severely underutilized potential.

List of Figures

Appendix A

Code Example for libxom

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include "libxom.h"

// This is the function we want to move into XOM
// We link it into the .data section, so that we can overwrite it later
// Note that in a real-world application, such functions are typically
// created at runtime
unsigned int __attribute__((section(".data")))
    secret_encryption_function (unsigned int plain_text) {
    // Just XOR to keep it simple
    return plain_text ^ 0xcafebabe;
}
void __attribute__((section(".data"))) secret_encryption_function_end (void) {}


int main(int argc, char* argv[]) {
    // Get the secret function's size
    const size_t secret_function_size =
                (size_t) secret_encryption_function_end -
                (size_t) secret_encryption_function;

    unsigned int (*secret_function_xom)(unsigned int);
    int status;

    // 'struct xombuf' is an anonymous struct representing one or more XOM pages
    struct xombuf* xbuf;

    unsigned int plain_text = 0xdeadbeef;
    unsigned int cipher_text;

    // Abort if XOM is not supported
    if(get_xom_mode() == XOM_MODE_UNSUPPORTED)
        return 1;
```

```
36      // Allocate a XOM buffer consisting of a single page
37      xbuf = xom_alloc(PAGE_SIZE);
38      if(!xbuf)
39          return errno;
40
41      // Write the secret function into the XOM buffer at offset 0
42      status = xom_write(xbuf, secret_encryption_function, secret_function_size, 0);
43      if(status <= 0)
44          return errno;
45
46      // Lock the XOM buffer, function returns a pointer to the XOM page itself
47      secret_function_xom = xom_lock(xbuf);
48      if(!secret_function_xom)
49          return errno;
50
51      // Overwrite the original function, key is now purged from readable memory
52      memset(secret_encryption_function, 0, secret_function_size);
53
54      if(get_xom_mode() == XOM_MODE_SLAT) {
55          // Mark the page for full register clearing if supported
56          // The second parameter can be set to 0 for vector-register clearing
57          status = xom_mark_register_clear(xbuf, 1, 0);
58          if (status < 0)
59              return -status;
60      }
61
62      // Call the function in XOM
63      // The following block restarts when full register clearing occurs
64      expect_full_register_clear {
65          cipher_text = secret_function_xom(plain_text);
66      }
67
68      printf("Cipher Text: 0x%x\n", cipher_text);
69
70      // Free the XOM buffer
71      xom_free(xbuf);
72
73      return 0;
74  }
```

The above example is a demo program using libxom, written in C. It first allocates a XOM buffer, fills it with the code of a secret function, and locks it. The XOM buffer is then marked for full register clearing if supported in the current environment. Note that the function in XOM is called inside of an `expect_full_register_clear` block, which starts from the beginning when full register clearing occurs.

Bibliography

[1] M. Schink and J. Obermaier, "Taking a look into Execute-Only memory," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/schink

[2] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1342–1353.

[3] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 763–780.

[4] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, 2015, pp. 325–336.

[5] S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "Exoshim: Preventing memory disclosure using execute-only kernel code," in *Proceedings of the 11th International Conference on Cyber Warfare and Security*, 2016, pp. 56–66.

[6] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "krˆ x: Comprehensive kernel protection against just-in-time code reuse," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 420–436.

[7] S. Gravani, M. Hedayati, J. Criswell, and M. L. Scott, "Iskios: Lightweight defense against kernel-level code-reuse attacks," *arXiv preprint arXiv:1903.04654*, 2019.

[8] F. Berlakovich and S. Brunthaler, "R2c: Aocr-resilient diversity with reactive and reflective camouflage," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 488–504.

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, Intel Corporation, March 2023, order number 325384-079US.

[10] F. J. Corbató and V. A. Vyssotsky, "Introduction and overview of the multics system," in *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, 1965, pp. 185–196.

[11] C. Yarvin, R. Bukowski, and T. Anderson, "Anonymous rpc: Low-latency protection in a 64-bit address space." in *USENIX Summer*, 1993, pp. 175–186.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *Acm Sigplan Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[13] S. Gueron, "Memory encryption for general-purpose processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[14] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper*, p. 13, 2016.

[15] D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, "uxom: Efficient execute-only memory on armcortex-m," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 231–247.

[16] Z. Shen, K. Dharsee, and J. Criswell, "Fast execute-only memory for embedded systems," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 7–14.

[17] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 574–588.

[18] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 227–242.

[19] *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Advanced Micro Devices Inc., June 2023, 24593 - Rev.3.41.

[20] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 489–504.

[21] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "Erim: Secure, efficient in-process isolation with protection keys (mpk)," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1221–1238.

[22] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, "You shall not (by) pass! practical, secure, and fast pku-based sandboxing," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 266–282.

[23] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "Pku pitfalls: Attacks on pku-based memory isolation systems," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1409–1426.

[24] F. Rodríguez-Haro, F. Freitag, L. Navarro, E. Hernánchez-sánchez, N. Farías-Mendoza, J. A. Guerrero-Ibáñez, and A. González-Potes, "A summary of virtualization techniques," *Procedia Technology*, vol. 3, pp. 267–272, 2012.

[25] M. Belopuhov, "Implementation of xen pvhvm drivers in openbsd," 2016.

[26] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Minix 3: A highly reliable, self-repairing operating system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 80–89, 2006.

[27] Google Inc., "Fuchsia os f14." [Online]. Available: https://fuchsia.dev/

[28] S. K. Tesfatsion, C. Klein, and J. Tordsson, "Virtualization techniques compared: Performance, resource, and power usage overheads in clouds," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 145–156. [Online]. Available: https://doi.org/10.1145/3184407.3184414

[29] The Linux Foundation, Citrix Systems Inc., "Xen v.4.18." [Online]. Available: https://xenproject.org

[30] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel Corporation, September 2023, order number 253665-081US.

[31] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*, Intel Corporation, March 2023, order number 325383-079US.

[32] S. Gueron and M. E. Kounavis, "Intel® carry-less multiplication instruction and its usage for computing the gcm mode," *White Paper*, p. 10, 2010.

[33] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting mobile communications: The insecurity of 802.11," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, 2001, pp. 180–189.

[34] Intel Corporation, "Retpoline: A branch target injection mitigation," June 2018.

[35] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corporation, January 2023, order number 248966-046A.

[36] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[37] L. Torvalds, "Linux v6.6.8, commit id `4c9646a796d66a2d81871a694e88e19a38b115a7`," https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git?h=v6.6.8, 2023.

[38] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1871–1885.

[39] D. P. Bovet, "Implementing virtual system calls," *LWN.net*, October 2014, https://lwn.net/Articles/615809/ (Retrieved 2024-01-12).

[40] U. Drepper, "How to write shared libraries," December 2011.

[41] Free Software Foundation Inc., "The GNU C Library v2.38." [Online]. Available: https://sourceware.org/glibc/libc.html

[42] Android OS Documentation, "Execute-only Memory (XOM) for AArch64 Binaries," March 2024, https://source.android.com/docs/security/test/execute-only-memory (Retrieved 2024-03-24).

[43] Igor Sysoev, F5 Inc., "nginx v.1.23.2." [Online]. Available: https://nginx.org

[44] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[45] Michael Larabel, "Phoronix Test Suite Nginx Benchmark v.3.0.1." [Online]. Available: https://openbenchmarking.org/test/pts/nginx

[46] M. Larabel, "Looking at the linux performance two years after spectre / meltdown mitigations," *Phoronix.com*, January 2020, https://www.phoronix.com/review/spectre-meltdown-2 (Retrieved 2024-02-28).

[47] *ARM Architecture Reference Manual, Thumb-2 Supplement*, ARM Limited, December 2005, dDI 0308D.

[48] L. De Moura and N. Bjørner, "Z3: an efficient smt solver," in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[49] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, "Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals," 2019.

[50] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[51] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert, "Intel virtualization technology for directed i/o." *Intel technology journal*, vol. 10, no. 3, 2006.

[52] I. AMD and O. Virtualization, "Technology (iommu) specification," 2007.

[53] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[54] G. Maisuradze and C. Rossow, "ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2109–2122. [Online]. Available: https://doi.org/10.1145/3243734.3243761

[55] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 249–266.

[56] J. Wikner and K. Razavi, "{RETBLEED}: Arbitrary speculative code execution with return instructions," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3825–3842.

[57] J. Corbet, "Stuffing the return stack buffer," *LWN.net*, July 2022, https://lwn.net/Articles/901834/ (Retrieved 2024-02-10).

[58] Intel Corporation, "Speculative Execution Side Channel Mitigations (Version 2.0)," May 2022, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html (Retrieved 2024-02-10).

[59] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018.

[60] *Intel Analysis of Speculative Execution Side Channels*, Intel Corporation, May 2018, document number 336983-004.

[61] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith, "Ghostbusting: Mitigating spectre with intraprocess memory isolation," in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, 2020, pp. 1–11.

[62] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[63] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[64] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.

[65] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 769–784.

[66] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," *arXiv preprint arXiv:1806.07480*, 2018.

[67] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1852–1867.

[68] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, "Port contention for fun and profit," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 870–887.

[69] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *ASPLOS*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 673–686. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304062

[70] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. Dray Jr, "Advanced encryption standard (aes)," 2001.

[71] D. J. Bernstein *et al.*, "Chacha, a variant of salsa20," in *Workshop record of SASC*, vol. 8, no. 1. Citeseer, 2008, pp. 3–5.

[72] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-bit block cipher suitable for multiple platforms—design andanalysis," in *Selected Areas in Cryptography: 7th Annual International Workshop, SAC 2000 Waterloo, Ontario, Canada, August 14–15, 2000 Proceedings 7*. Springer, 2001, pp. 39–56.

[73] D. Kwon, J. Kim, S. Park, S. H. Sung, Y. Sohn, J. H. Song, Y. Yeom, E.-J. Yoon, S. Lee, J. Lee, S. Chee, D. Han, and J. Hong, "New block cipher: Aria," in *Information Security and Cryptology - ICISC 2003*, J.-I. Lim and D.-H. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 432–445.

[74] OpenSSL Project Authors, "Openssl v.3.2.1." [Online]. Available: https://www.openssl.org/

[75] Free Software Foundation, Inc., g10 Code GmbH, "libgcrypt, version 1.10.2," April 2023. [Online]. Available: https://gnupg.org/software/libgcrypt/

[76] T. Hansen and D. E. E. 3rd, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)," RFC 6234, May 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6234

[77] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2104

[78] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "Intel sha extensions–new instructions supporting the secure hash algorithm on intel architecture processor," *Intel White Paper*, 2013.

[79] M. Larabel, "In light of spectre bhi, the performance impact for retpolines on modern intel cpus," *Phoronix.com*, March 2022, https://www.phoronix.com/review/spectre-bhi-retpoline (Retrieved 2024-02-28).

[80] G. Patat, M. Sabt, and P.-A. Fouque, "Exploring widevine for fun and profit," in *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022, pp. 277–288.

[81] Microsoft Corporation, "WriteProcessMemory function documentation," (Accessed 2024-02-12). [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-writeprocessmemory

[82] A. Maario, V. K. Shukla, A. Ambikapathy, and P. Sharma, "Redefining the risks of kernel-level anti-cheat in online gaming," in *2021 8th International Conference on Signal Processing and Integrated Networks (SPIN)*, 2021, pp. 676–680.

[83] National Security Information, "Cybersecurity information: Enforce signed software execution policies," August 2019, (Accessed 2024-02-12). [Online]. Available: https://media.defense.gov/2019/Sep/09/2002180334/-1/-1/0/Enforce%20Signed%20Software%20Execution%20Policies%20-%20Copy.pdf

[84] A. Ionescu, "SimpleVisor," 2018, Commit ID `989d33b1bc6569965d7aad3bd50a8d35fa4c359e`. [Online]. Available: https://github.com/ionescu007/SimpleVisor.git

[85] "SiSyPHuS Win10, Work Package 6: Virtual SecureMode," Federal Office for Information Security, Tech. Rep., 03 2019.

[86] Siguza, "PAN," January 2020, https://blog.siguza.net/PAN/ (Retrieved 2024-03-24).

[87] S. Sparks and J. Butler, "Shadow walker: Raising the bar for rootkit detection," *Black Hat Japan*, vol. 11, no. 63, pp. 504–533, 2005.

[88] J. Torrey, "More shadow walker: Tlb-splitting on modern x86," *Blackhat USA*, 2014.

[89] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 35–46.

[90] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 256–267.

[91] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of jit-rop defenses," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 139–156.

[92] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, S. Crane, C. Liebchen, P. Larsen, L. Davi, M. Franz *et al.*, "Address oblivious code reuse: On the effectiveness of leakage resilient diversity." in *NDSS*, 2017.

[93] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[94] B. Garmany and T. Müller, "Prime: private rsa infrastructure for memory-less encryption," in *Proceedings of the 29th Annual Computer Security Applications Conference*, ser. ACSAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 149–158. [Online]. Available: https://doi.org/10.1145/2523649.2523656

[95] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without ram." in *NDSS*, 2014, pp. 23–26.

[96] C. Li, L. Guan, J. Lin, B. Luo, Q. Cai, J. Jing, and J. Wang, "Mimosa: Protecting private keys against memory disclosure attacks using hardware transactional memory," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 3, pp. 1196–1213, 2021.

[97] Y. Yang, Z. Guan, Z. Liu, and Z. Chen, "Protecting elliptic curve cryptography against memory disclosure attacks," in *Information and Communications Security*, L. C. K. Hui, S. H. Qing, E. Shi, and S. M. Yiu, Eds. Cham: Springer International Publishing, 2015, pp. 49–60.

[98] Y. Fu, J. Lin, D. Feng, W. Wang, M. Wang, and W. Wang, "Regkey: A register-based implementation of ecc signature algorithms against one-shot memory disclosure," *ACM Trans. Embed. Comput. Syst.*, vol. 22, no. 6, nov 2023. [Online]. Available: https://doi.org/10.1145/3604805