Lixom: Protecting Encryption Keys with Execute-Only Memory

Tristan Hornetz, Lukas Gerlach, and Michael Schwarz

CISPA Helmholtz Center for Information Security
`<firstname>.<lastname>@cispa.de`

Abstract. The confidentiality of cryptographic secrets is crucial for the security of modern computing systems. However, ensuring confidentiality is difficult in the presence of privileged attackers or transient-execution vulnerabilities such as Meltdown or Spectre. While Trusted Execution Environments (TEEs) provide robust protection, they suffer from limited hardware availability, performance overhead, and the need for substantial system redesign, making them impractical for many deployments.
In this paper, we present Lixom, a lightweight and generic technique to prevent data leakage of cryptographic secrets on x86 processors. Lixom achieves its confidentiality guarantees by storing secrets in code instead of data and preventing access to them with execute-only memory (XOM). In virtual machines, Lixom protects secrets from a compromised guest kernel, providing security guarantees akin to TEEs. Additionally, Lixom protects against Spectre, Meltdown, and Foreshadow attacks without performance overhead for algorithms such as AES. In 3 case studies, we show that Lixom improves the security of applications like disk encryption or digital rights management in real-world applications.

1 Introduction

Cryptographic secrets are high-value targets that require special protection from attackers. However, protecting cryptographic secrets is incredibly challenging if attackers have native privileged code execution, i.e., root attackers, or can exploit transient-execution attacks [9], such as Spectre [20] or Meltdown [25]. Existing hardware mechanisms, such as Trusted Execution Environments (TEEs) or Trusted Platform Modules (TPMs), can protect secrets against many such attacks [10], but they are not universally available. Moreover, even TEEs are susceptible to microarchitectural attacks breaking confidentiality [45, 11, 29, 24, 5, 52]. Additionally, many systems still use hardware vulnerable to powerful and virtually unfixable transient-execution attacks such as Meltdown [25] or Foreshadow [45] that leak data across security domains.

This paper presents Lixom, a generic protection mechanism that prevents the disclosure of cryptographic secrets by leveraging *Execute-only memory (XOM)* on x86. The core idea of Lixom is to embed secret data directly into code, such as encoding secrets as immediate values in `mov` instructions, rather than storing them in data sections. XOM then prevents any direct disclosure or modification of these secrets while permitting their regular usage by executing the protected code. This provides leakage resistance and helps with policy enforcement, as secrets are only usable within the scope of what the protected code allows.

Lixom considers a powerful threat model: We show that it can protect secrets from attackers that have fully compromised the kernel of a VM guest, a level of protection previously only possible with a TEE. We achieve this by utilizing Intel's hypervisor-controlled *Extended Page Tables (EPT)* [18] to enforce XOM. Furthermore, we introduce two novel techniques: *Page locking*, which allows guests to securely relinquish read access to sensitive code, and *register clearing*, which prevents a malicious guest kernel from disclosing a program's register state. Together, these measures can defend against many attacks, including transient-execution attacks. The reason is one observation we make in this paper: Most transient-execution attacks only leak data, not code. We demonstrate this fact for transient-execution attacks, such as Meltdown [25] or Foreshadow [45].

For native, non-virtualized environments, we additionally present Lixom-Light, which uses *Memory Protection Keys (MPK)* instead of EPT to enforce XOM [18]. This hardware mechanism is widely available and well-supported in the Linux kernel. Lixom-Light provides strong disclosure protection against transient-execution attacks and is practical to deploy on existing software stacks.

We evaluate Lixom with 3 case studies in which we protect cryptographic algorithms and analyze the security benefits and performance overhead of Lixom. Our case studies involve the secure handling of password hashes and key protection for AES and HMAC. Furthermore, we integrate these implementations into an OpenSSL provider module, making them available to established programs and tools, such as the nginx web server. As Lixom involves hypervisor-based components, we create a customized version of the Xen hypervisor and a set of libraries that make EPT available to guests securely. Our results show that Lixom provides strong disclosure protection without affecting the throughput of AES and outperforming OpenSSL by up to 6 %, making Lixom an ideal hardening mechanism for applications like disk encryption or digital rights management.

Contributions. In summary, the contributions of this paper are:

1. We propose and design Lixom, a technique to protect cryptographic secrets with execute-only memory. Lixom works without hardware changes.
2. We create implementations for cryptographic algorithms that use Lixom, demonstrating their practicability.
3. We show that Lixom is resistant against various attack vectors, including transient-execution attacks, and can defend against privileged attackers.
4. We perform an extensive performance analysis to show the low runtime overhead of Lixom.

Availability. Lixom is available at `https://github.com/cispa/Lixom`.

2 Background

In the following, we introduce the necessary background to understand the remainder of this paper.

2.1 Memory Protection Keys

Execute-only memory (XOM) permits instruction fetches but can neither be read nor written. Although simple in principle, the x86 page tables do not support XOM, which means that programs must use alternative mechanisms to create XOM mappings [18]. One method for XOM are Memory Protection Keys (MPK), which are widely supported on recent hardware [18]. This feature allows tagging page table entries with a 4-bit protection key, with each possible value associated with a configurable set of access restrictions. Programs can then modify these restrictions in the special 32-bit *PKRU* register. Notably, this does not require supervisor-mode privileges, making it trivial to turn off their restrictions. It is, therefore, challenging to use them for security purposes, with early works containing various vulnerabilities [44, 17, 47]. Nevertheless, MPK sees use in certain sandboxing techniques [44, 17, 47] and as a Spectre mitigation [19], demonstrating that this feature can provide security benefits if used correctly.

2.2 Hardware-assisted Virtualization and EPT

Virtualization allows the execution of multiple *guest* operating systems on a single *host* machine. The *hypervisor* occupies the highest privilege domain and governs hardware resources and communication between guests. Modern CPUs support virtualization in hardware, e.g., as Intel's *virtual-machine extensions (VMX)* [18, 2]. VMX introduces a mode of operation for guests named *VMX non-root operation*, which, from a guest perspective, is indistinguishable from native execution. However, privileged instructions, interrupts, or accesses to protected memory regions can trigger *VM exits*, which transfer control from the guest to the hypervisor. Another addition of VMX is *Second Layer Address Translation (SLAT)*. With SLAT, guest-managed page tables translate virtual addresses to so-called *guest-physical addresses*. These addresses undergo a second translation step with the hypervisor-managed *Extended Page Tables (EPT)*, resulting in a hardware-backed physical address. Notably, however, EPT entries follow a different format than the regular page tables on x86, allowing for the creation of XOM. EPT-enforced XOM provides security guarantees that are far stronger than MPK. Moreover, since the hypervisor manages EPT entries, not even guest kernels can modify them, minimizing the attack surface. This approach to XOM was pioneered by Readactor [12], which protects diversified code with EPT to prevent code-reuse attacks. Other works utilizing EPT-enforced XOM include KHide [14] and ExOShim [7]. Unfortunately, EPT is exclusive to Intel CPUs, making approaches involving it incompatible with AMD. While AMD's Reverse Map Table (RMP) provides a similar mechanism to enforce XOM [2], this is far less widely supported, and not considered in the remainder of this paper.

3 Design of Lixom

This section presents the design of Lixom and Lixom-Light. Section 3.1 provides an overview, Section 3.2 defines the threat model, and Section 3.3 introduces the challenges we have to solve.

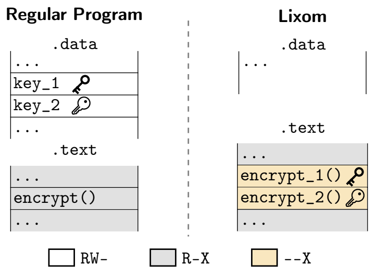

Fig. 1: Key storage in memory with Lixom

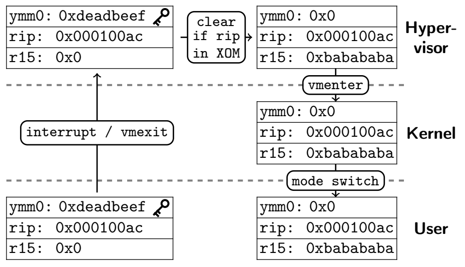

Fig. 2: Illustration of the register clearing process. *r15* is the signal register.

3.1 Overview

The main idea of Lixom is to move secrets, such as cryptographic key material, to code protected by XOM, as shown in Figure 1. This conversion can be as simple as having a `mov` instruction with an immediate value as the source operand. The hardware then prevents reading the secret both architecturally and transiently (e.g., via Meltdown). As Lixom relies on EPT-enforced XOM for the highest security guarantees, it also requires a method for securely allocating and deallocating such pages in the guest. We support two operations guests can invoke: *lock* and *unlock*. *lock* transforms regular memory pages into XOM, preventing read and write accesses. *unlock* transforms XOM pages into regular memory, but only after clearing them. This way, guests can dynamically allocate XOM without compromising secrets. We refer to this concept as *page locking*. However, while the main idea is intuitive, implementing it requires solving several challenges discussed in Section 3.3 and ultimately solved with Lixom.

3.2 Threat Model

For Lixom, we assume an attacker who fully compromised the kernel of a VM guest. They can remap, read, and overwrite memory, execute arbitrary code in the victim's context, and interrupt the victim at any time. The attacker can mount Spectre and other microarchitectural attacks. Fault injection attacks, e.g., Rowhammer [37], are out of scope. For Lixom-Light, the attack runs without kernel privileges and aims to disclose secrets from another security domain. We assume that the isolation is not compromised architecturally level. However, attackers can execute arbitrary code in their own unprivileged process.

3.3 Challenges

While the basic idea of Lixom is intuitive, we must address several challenges to ensure robust guarantees for the protection of cryptographic secrets.

Memory-less Cryptography. Most implementations of cryptographic algorithms store key-dependent data in regular memory accessible to a privileged attacker. Hence, code protected with Lixom must use *memory-less cryptography*, keeping secrets and intermediary results in the registers at all times. While the capacity of the general-purpose register is generally too small to hold key material and perform meaningful computation simultaneously, the vector registers provide significantly more storage space. AVX2 provides registers with 512 bytes of confidential "memory", ample for many cryptographic algorithms. Previous works established this approach to protect encryption keys against cold-boot attacks [30]. If more memory is required, secrets are encrypted with AES-NI before saving them to memory. The AES-NI key is stored in code alongside other protected secrets [13, 15]. Additionally, we expect programs to authenticate such backups, for example with schemes like AES-GCM. Confidentiality and integrity are strictly required, as the attacker has full control over non-XOM memory.

Defending against Interrupts. XOM guarantees protection for secrets in memory. However, at some point, confidential information must enter the register state for processing. If a privileged attacker interrupts the program at such times, they can disclose any secrets that are currently in use. We call code sections where secrets are in the registers *critical zones* (not to be confused with critical sections in concurrent programming). While it is possible to restrict how the guest can interrupt a program, non-maskable interrupts (NMIs) cannot be turned off or deferred. Should an NMI occur by chance at the wrong time, it could compromise encryption keys. With Lixom, we do not attempt to prevent interrupts. Instead, we leverage that VMX permits hypervisors to handle interrupts before transferring control back to the guest. This allows us to perform *register clearing*, where the hypervisor partially overwrites the register state when handling interrupts occurring during the execution of an XOM page. In practice, Lixom only overwrites the processor's vector registers and two general-purpose registers, making it easy to recover from clearing events, as the instruction pointer and most general-purpose registers remain unaffected. At the same time, programs can store and process key material in the vector registers. One of the 2 overwritten general-purpose registers transfers 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. Figure 2 illustrates register clearing and how different privilege levels perceive the program's register state during a clearing event.

Defending against Control-Flow Hijacking. Even with the aforementioned measures in place, there remains a potent attack vector against programs in XOM. An attacker may hijack control flow in a critical zone and redirect it to a disclosure primitive. The easiest way to achieve this is by manipulating code pointers in memory, such as return or function pointers. Spectre attacks, which *speculatively* redirect control flow, also pose a significant risk. The solution we propose to this problem is the complete elimination of indirect branches in critical zones. As shown in previous work [43, 4], this elimination can be done automatically. This way, there are no code pointers to manipulate, and speculative branch target injection with Spectre is no longer possible. Furthermore, while Spectre can still manipulate direct conditional branches, the branch targets are under the programmer's control. Thus, it is possible to ensure that speculative control flow cannot exit the critical zone without going through designated 'exit points' that erase any secrets from the register state. Without gadgets in the critical zone, Spectre attacks are prevented.

Apart from pointer manipulation and Spectre, there is a second way in which a privileged attacker could hijack control flow: Although they cannot access the XOM pages directly, they can map them to an arbitrary location in a process's virtual address space. Therefore, if a program crosses a code page boundary in a critical zone, the next page can be any page chosen by the guest kernel. The kernel can thus disclose the register state by inserting gadget code there. Unfortunately, this limits the maximum size of a critical zone to a single code page. More complex programs may require 2 MB pages instead of 4 kB pages.

Finally, an attacker may manipulate control flow by priming the registers with chosen values and jumping to arbitrary addresses in critical zones. This is usually not a problem, as a critical zone typically starts with the secret being loaded into the register state. If the start is not executed, the secrets are not in the registers and, hence, cannot be leaked. However, in cases where the program loads additional secrets later, leakage must be avoided with defensive programming measures, such as assertions on the current state.

3.4 Lixom-Light

Lixom-Light employs all programming rules of Lixom, but uses MPK instead of EPT to enforce XOM. It does not rely on virtualization and hence uses neither page locking nor register clearing. Since it does not rely on EPT, it is compatible with both Intel and AMD.

In contrast to Lixom, Lixom-Light only considers attackers with code execution in a different privilege domain. If isolation is not compromised on an architectural level, this attacker must resort to transient execution attacks, which exploit flaws in the microarchitecture. We argue that in this scenario, we can safely rely on MPK. To disable its XOM protection, the attacker must execute code architecturally, which is not the case, e.g., with Spectre attacks. Furthermore, the `wrpkru` instruction, which is the only way to modify PKRU, serializes memory accesses similarly to memory fences [18]. Therefore, Spectre attackers may execute a `wrpkru` gadget but cannot leak code afterward. Lixom-Light hence provides robust disclosure protection against Spectre. In Section 6.2, we show that Lixom-Light also resists attacks like Meltdown and RIDL [46], making it a practical defense against several classes of transient execution attacks.

4 Implementation

We develop a proof-of-concept implementation of Lixom via a series of patches for the Xen hypervisor (Section 4.1), a Linux kernel module named *modxom* (Section 4.2), and a user-mode library called *libxom* (Section 4.3).

4.1 Xen Hypervisor Patches

We extend the Xen hypervisor to support page locking and register clearing.

Page Locking. Our patches add a *vmcall* interface for locking 4 kB memory pages into XOM. Guests can then enable register clearing for individual XOM pages. Furthermore, our patches add a *sub-page XOM* mechanism, which allows guests to manage XOM on a 128 B granularity. This makes Lixom more memory efficient since programs would otherwise have to lock an entire 4 kB for every individual secret, thus wasting memory if the protected code is smaller. Setting up a memory range for sub-page XOM zeroes it, and immediately locks it into XOM without initialization. However, guests can invoke the hypervisor to populate uninitialized sub-pages with data, with the hypervisor ensuring that a sub-page can only be written to when still uninitialized. Once the data is in place, it is therefore unreadable and immutable, as with regular XOM. Unlocking uses the regular page-granularity *unlock* function, freeing all sub-pages at once. This way, we can support page locking for memory ranges smaller than 4 kB.

Register Clearing. We enforce the register clearing mechanism in Xen's VM exit handler. Whenever an interrupt or a fault occurs, we check whether the currently executed code page is among the previously marked XOM pages and modify the guest's register state accordingly. To reliably catch every interrupt, our patches also disable the virtual interrupt controller (vAPIC) of VMX by default, as it typically delivers interrupts directly to the guest kernel rather than the hypervisor. While enabling the vAPIC is technically still possible, we strongly discourage it, as this may undermine the security of register clearing.

4.2 Kernel Module

modxom, Lixom's kernel module, serves two functions: Firstly, it provides an interface to issue hypervisor operations from user-space, as user-mode programs cannot issue hypercalls directly. Secondly, *modxom* prevents the Linux kernel from attempting to read from XOM pages, which could lead to irrecoverable error conditions, i.e., when swapping out or reusing pages. To effectively address these issues, modxom implements a separate in-kernel memory allocator and restricts the hypervisor operations to memory allocated with this mechanism. User-mode programs can access this allocator through the *mmap* system call, and issue operations with *write* calls to a special file in Linux's `/proc` filesystem. Through this approach, modxom reliably ensures that every XOM page is pinned to memory, thereby preventing it from being swapped out. Additionally, modxom's handler for the *close* system call is invoked whenever a process closes its handle to the `/proc` file, which occurs when the process terminates or crashes. Therefore, we can reliably unlock every XOM page that is still in use at this point, preventing Linux from reusing locked XOM pages.

4.3 User Space Library

Finally, we present a shared user-mode library that simplifies the management of XOM by abstracting modxom's interface. Additionally, the user-space library can emulate the Xen hypervisor's XOM behavior with MPK if the hardware supports it. This way, programs setting up a Lixom environment do not need to distinguish between Lixom-Light and Lixom, with any code for Lixom also working for Lixom-Light. We use this mechanism for our case studies so that all experiments for both Lixom-Light and Lixom can use the same code.

5 Case Studies

In this section, we present 3 case studies in which we implement programs that utilize Lixom to protect cryptographic secrets. Section 6 evaluates the performance of these implementations.

5.1 Case Study 1: Message Authentication with HMAC

We show that it is possible to perform message authentication with HMAC-SHA256 purely in XOM using Lixom. Although widely supported hardware extensions exist for SHA-256, they are only partially usable as the SHA extensions only cover certain primitive operations. Therefore, the bulk of SHA-256 must be implemented using more conventional techniques. These techniques involve control-flow structures that are challenging to create using only direct branches. Our case study, therefore, provides insights into the performance of Lixom for more involved algorithms.

An implementation challenge of SHA-256 is that it uses round constants, manipulation of which may undermine the algorithm's security guarantees. To prevent attackers from changing these round constants, we store them in code as immediate values. As with the key, each round constant requires a small code segment moving it to the correct register. Although these segments are not large individually, they alone consume 720 bytes of the available 4 kB memory page.

Another challenge is that the hash state, which updates with each message block, is lost after register clearing and cannot be easily recovered. Therefore, we utilize authenticated encryption with AES-128-GCM to confidentially store the hash state in memory at regular intervals. When interrupted, the hash function then restores its internal state from the latest checkpoint instead of starting from scratch. This enables the authentication of messages of arbitrary size, even with heavy CPU contention from other processes. The encryption key is generated randomly and inserted into the program simultaneously with the HMAC key.

5.2 Case Study 2: Encryption with AES

Our second application for Lixom protects AES encryption keys. The AES-NI instruction set extensions allow for implementing AES with little to no control flow structures. Furthermore, most AES-NI instructions work on vector register operands, making it easy to derive round keys and perform encryptions without writing key material to memory. For most modes of operation, gracefully handling register clearing events is relatively straightforward, as the program can check the signal register after encrypting a message block. When the registers are cleared, we simply need to re-derive the round keys, and can continue at the block that was last processed. The current block offset is not confidential and is thus stored in a register that is unaffected by register clearing.

For this paper, we provide two Lixom-compliant implementations of AES-128-CTR: One utilizing the 128-bit AES-NI extensions and one using the 256-bit VAES extensions. Both implementations can also serve as the GCTR function for AES-128-GCM, allowing for authenticated encryption with Lixom. Our case study does not pre-compute the round keys, as loading them from immediates takes roughly 330 bytes of code under Lixom's rules, whereas key expansion with AES-NI takes as little as 211 bytes. An implementation with pre-computed round keys may exhibit slightly better runtime performance at a higher memory cost per encryption key.

5.3 Case Study 3: Protecting Password Hashes

One of the more straightforward application scenarios for Lixom is to provide leakage resistance to password hashes, which are popular targets for Spectre attacks [49, 41]. Following the programming rules from Section 3 in this scenario is trivial, as the correct hash is subject only to simple equality checks. If implemented with conditional move instructions, this does not require any branches. Therefore, once the protected code is fully initialized, there is no longer any attack surface for Spectre. Note that this specific application does not benefit from Lixom's resistance against privileged attackers, as the guest kernel can trivially disclose the correct hash when a user authenticates with the correct password. Attackers may also use the protected code as an oracle for dictionary attacks, albeit at a relatively low test rate compared to the GPU-powered methods available when the hash is known directly. Nevertheless, we argue that Lixom's low cost in this scenario justifies its use as a hardening technique.

6 Evaluation of Lixom

In this section, we evaluate the performance and attack resistance of Lixom.

6.1 Performance

Our performance study of Lixom investigates 3 aspects: Encryption throughput, setup costs, and application performance. We analyze the former two aspects with custom benchmarks and employ the nginx benchmark of the Phoronix test suite [22] to gauge Lixom's impact on real-world applications. All tests utilize a custom-built OpenSSL provider library, making the case-study implementations from Section 5 available to any program using OpenSSL without requiring code changes. This way, the nginx benchmark utilizes the Lixom-compliant AES-128-GCM implementation for its TLS connections. In summary, the only significant overhead of Lixom occurs when setting up an encryption context. However,

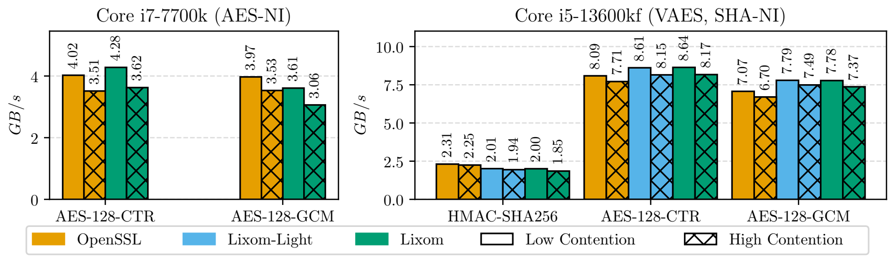

Fig. 3: Throughput of our Lixom implementations ($n = 128$, $\frac{SE}{\mu} < 0.7\,\%$, 256 MB per sample, single thread). Cross-hatching indicates that the test was conducted with high contention for CPU time, causing frequent interrupts.

some of our implementations outperform OpenSSL in throughput, meaning that Lixom may not incur any overhead, depending on the application.

Test Environment. Unless otherwise stated, we test Lixom on a 2-core HVM guest of our modified Xen hypervisor based on Xen v4.18.1. The guest runs Debian 12 with Linux v6.1.0. Tests involving OpenSSL use OpenSSL v3.0.11.

Data Throughput. The first benchmark series measures the data throughput of our Lixom-compliant implementations under various conditions and hardware configurations. Furthermore, it aims to quantify to which extent frequent interrupts degrade the performance of Lixom, since the protected code must run recovery procedures after register clearing. We run two threads parallel to the benchmark to increase the interrupt frequency, one with high CPU usage and one spinning the *sync* system call. On the test VM with two virtual cores, this fully utilizes the available CPU resources, while the frequent system calls require many context switches. Note that the benchmarks for Lixom-Light and Lixom execute the same code, only with a different XOM-enforcement method and the addition of register clearing for Lixom.

Figure 3 shows the results. All AES implementations except for AES-128-GCM on the Core i7 7700K slightly outperform OpenSSL. Monitoring the performance counters reveals this is primarily due to execution stalls when loading the plain text. On the Core i5 13600KF, for instance, the OpenSSL AES-128-CTR benchmark executes 80.4 % more stall cycles on average, and 161 % more stall cycles with a concurrently pending L3 cache miss. Contrarily, our HMAC implementation's throughput is 13.4 % lower than OpenSSL's. This is expected, as the code requires changes to eliminate indirect branches, load round constants from the code, and backup internal state.

While Lixom-Light is faster than Lixom on average, the overhead is low. The most significant difference occurs in the high-contention HMAC-SHA256 benchmark, where Lixom-Light's throughput is 4.9 % higher than Lixom's. This, too, is expected, as Lixom loses its progress up to the last checkpoint when interrupted, whereas Lixom-Light does not. However, there is virtually no overhead with the AES benchmarks. For AES-128-CTR, Lixom's average throughput is even 0.3 % higher than Lixom-Light's. We conclude that register clearing has a

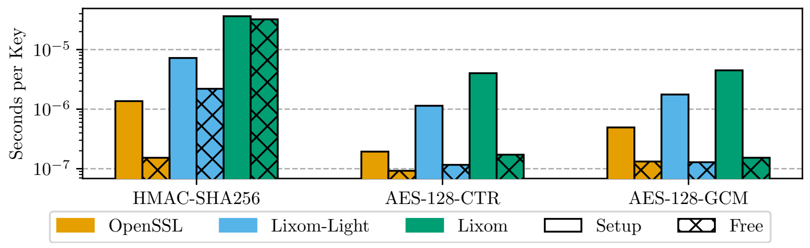

Fig. 4: Mean time required for allocating and freeing encryption contexts ($n = 2^{14}$, Intel Core i5 13600kf). Less is better.

minor impact on performance if the protected code section has appropriate recovery mechanisms. Note, however, that an algorithm more complex than HMAC or AES may require more involved recovery procedures, increasing performance degradation from register clearing.

Setup Costs. Another source of overhead are setup and teardown costs. Lixom requires creating and freeing a separate executable memory segment for every secret we use, meaning we need to modify the NX bit in the page tables. In contrast to standard memory allocations, this always requires a system call. For Lixom, creating and freeing XOM requires a hypercall on top of this. Figure 4 shows the overhead of the setup. Both Lixom-Light and Lixom have setup overheads that are higher than an unprotected OpenSSL version. However, using sub-page granularity XOM eliminates most of the freeing costs for AES. This way, system- and hypercalls are only necessary when all 128 B sub-pages in an XOM region are freed. For the AES benchmarks, we allocate and free the sub-page XOM ranges in 16 kB chunks, so we only need a hypercall after freeing all encryption contexts in this range. Our HMAC implementation does not use this mechanism, hence the significantly higher freeing costs. These results indicate that Lixom is better suited for applications with few, rarely changing encryption keys. Note that our results are an upper bound for Lixom's setup costs. There is still room for optimization.

Application Performance. Finally, we investigate the performance impact of Lixom on the popular nginx web server using the Phoronix test suite [22]. Its nginx benchmark measures the number of requests nginx can handle per second while using our AES-128-GCM implementation for TLS connections. We use this benchmark because nginx is a worst-case scenario given Lixom's cost profile in the previous experiments. Encryption keys are frequently exchanged, meaning that nginx should be among the applications most affected by the high setup costs. We expect other real-world applications to be less affected. For comparison, we also perform a benchmark with Gramine-SGX v1.7 [42], which allows running nginx inside an SGX enclave. For this SGX benchmark only, we use KVM (Linux v6.1.0) as the hypervisor, as Xen does not fully support SGX. This benchmark is performed on a Core i7 8700, as neither of the other processors supports SGX Launch Control, which Gramine-SGX requires.

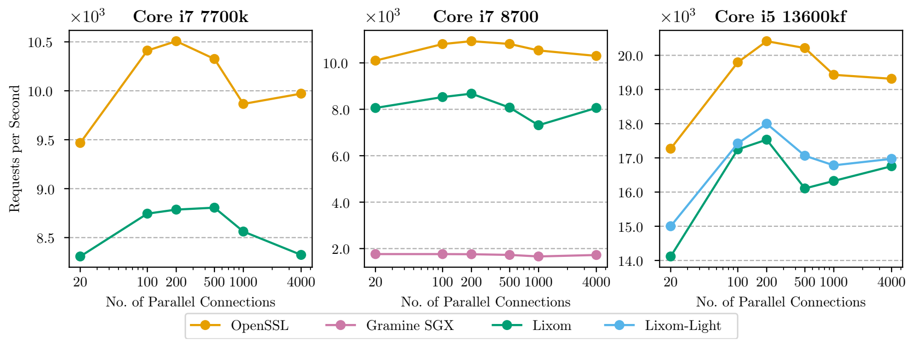

Fig. 5: Results from the Phoronix nginx benchmark v.3.0.1, which measures the number of HTTPS requests processed per second with parallel connections.

Figure 5 illustrates our results. As expected, Lixom reduces nginx's performance, with Lixom incurring a higher overhead than Lixom-Light. However, this overhead is significantly smaller than in the setup benchmark. Lixom-Light reduces the amount of requests per second by roughly 13 % on average across all configurations, and Lixom reduces them by roughly 18 %. Also, Lixom is significantly faster than Gramine-SGX, which reduces them by 83 %. This demonstrates that despite Lixom's high setup costs, the effect on real-world software is not as significant as in a raw benchmark, even in a worst-case scenario.

6.2 Attack Resistance

This section shows that Lixom mitigates many architectural and microarchitectural attacks, including transient execution attacks such as Spectre [20] and Meltdown [25]. We also discuss interrupt-based attacks [35] and DMA attacks.

Spectre-like attacks. We experimentally verify that XOM prohibits direct read access even under speculation and thus prevents attackers with Spectre read primitives from disclosing secrets in memory. The attack surface is, therefore, restricted to program sections where secrets are in the registers. Our programming rules effectively prevent the successful exploitation of these sections. Spectre-BTB [20] and Spectre-RSB [26] are impossible, and direct branches only target code within the protected code section. Furthermore, disclosure gadgets outside this section cannot be used since the program can only leave this section through exit points that erase secrets. If the protected code segments are free of exploitable Spectre gadgets, we can guarantee resistance against Spectre.

Other Transient Execution Attacks. Meltdown-type attacks exploit lazy exception handling in combination with out-of-order execution [9]. Such attacks have targeted various microarchitectural buffers, including the L1 data cache [25, 45], store buffer [8], load ports [46], and line-fill buffer [36]. However, none of these buffers hold any secret information in the context of Lixom. The programming rules prohibit storing secrets in memory, and loading secrets from memory uses instruction fetches, which affect none of these buffers directly. The only buffer in the core's memory subsystem that may hold instructions is the L2 cache, which, to our knowledge, is unaffected by any such attack. Therefore, attacks like Meltdown [25] and Foreshadow [48] do not threaten Lixom's confidentiality, even on affected processors. We experimentally verify this with an Intel Core i7 7700K processor. However, Lixom cannot protect secrets against attacks that target the vector registers, such as ZenBleed [31], Downfall [28], and RFDS [1]. These attacks are mitigated using microcode updates, making Lixom secure on these affected CPUs if the newest microcode is applied.

Side Channel Attacks. Side channels are a well-known threat to the confidentiality of cryptographic operations. As with any implementation of a cryptographic algorithm, we expect code for Lixom to be free from key-dependent control flow and other key-dependent memory access patterns. Such programming techniques prevent timing and cache-based side channels [32].

Interrupt-based attacks. Information gathering through interrupts is one of the most potent attack vectors against XOM. Naturally, this affects key material stored in the registers, which a privileged attacker can disclose with a well-timed interrupt. We argue that Lixom's register clearing fully prevents the leakage of cryptographic secrets through malicious interrupts. However, an attacker can still infer instruction semantics by observing changes in unprotected registers. Previous work demonstrates the practicability of similar code-recovery attacks [35]. We stress that the goal of Lixom is to protect cryptographic secrets, not to prevent code disclosure.

DMA attacks. Attacks leveraging peripheral devices with Direct Memory Access (DMA) bypass the processor's MMU and, thus, the page table configuration entirely. However, we can mitigate DMA attacks using an IOMMU, such as Intel's VT-d [18]. Our implementation of Lixom already considers this, preventing DMA peripherals from accessing XOM regions.

7 Discussion

7.1 Deployment and Potential Applications

Lixom is generic and applicable to a wide range of use cases. Therefore, the most practical means of deploying Lixom is in the form of a cryptographic library next to a set of hypervisor patches. While users can also integrate Lixom-compliant cryptography into projects directly, this necessitates assembly programming. Future work may explore the code generation for Lixom with a compiler pass.

Digital Rights Management (DRM) systems enforcing the copyright associated with remotely distributed media is another use case for Lixom. DRM systems encrypt media before delivery and only allow decryption in a restricted environment to prevent leakage of the encryption key. Modern DRM systems such as Google's Widevine rely on a TEE for this purpose [33]. However, many consumer-level x86 processors are not equipped with a TEE, forcing DRM systems to utilize obfuscation-based software-only mechanisms instead [33].

We argue that Lixom can improve upon pure obfuscation. With a TPM, it is possible to remotely attest the hypervisor's integrity to media distributors. If keys are exchanged with the hypervisor and then made available to guests with Lixom, its use is governed by a trusted component without reducing performance. This is generic and does not require the entire DRM system to be implemented in the hypervisor. Furthermore, since operating systems like Windows already use virtualization for security [27], we expect that this is easy to integrate.

Furthermore, Lixom can also efficiently allocate and manage EPT-enforced XOM for other purposes. While the primary functionality of Lixom is to hide cryptographic secrets, we envision our implementation's potential for further research involving XOM. We make our code publicly available, hoping it will facilitate future developments in this field. For example, it may help implement a leakage-resistant diversity scheme akin to Readactor [12].

7.2 Related Work

Other Ways to Create Execute-only Memory. While Lixom relies on explicit hardware support for XOM, other ways to enforce execute-only permissions exist. For instance, Sparks and Butler propose ShadowWalker [39], a technique to hide kernel rootkits via a *split TLB*. The instruction TLB's state differs from that of the data TLB through a special page fault handler, which yields different address translations depending on the type of access. However, shared higher-level TLBs make a split TLB unreliable on recent hardware [40]. Execute-no-Read (XnR) [3] uses a custom page-fault handler to keep only a sliding window of the most recently used code pages readable. Kwon et al. propose uXOM [21], which leverages *unprivileged memory instructions* on ARM Cortex-M. Such instructions always perform unprivileged memory accesses, regardless of the program's actual privileges. Hence, a privileged program with only unprivileged memory instructions can execute privileged code but not read it. Finally, approaches like LR^2 [6] and kRˆX [34] require only the standard $W \oplus X$ policy to enforce XOM, using range checks and software-based load address masking.

Microarchitectural Defense Mechanisms. T-SGX [38] defends against controlled-channel attacks on SGX using Intel TSX. T-SGX assumes a TEE-specific attack model in which the enclave is trusted, but the operating system is not. Most closely related to our work, Guan et al. [16] protect encryption keys by performing AES encryptions inside a TSX transaction. However, this approach is limited by the TSX transaction length.

Memory-less Encryption. Programs for Lixom must rely on memory-less encryption, where any key-dependent information remains in the registers. This concept was pioneered by TRESOR [30], which stores AES keys in the debug registers to defend against cold boot attacks. Later works, such as PRIME [13], Copker [15] and Mimosa [23] demonstrate the practicability of memory-less encryption for RSA. Yang et al. propose a memory-less implementation of ECDH key exchange algorithm with curve SECT163K1 [50].

Cryptographic Secrets in Protected Code. As part of uXOM, Kwon et al. proposed embedding encryption keys into XOM on ARM Cortex-M [21]. uXOM's threat model does not include transient execution or a malicious kernel, and the approach, therefore, lacks Lixom's defense measures. Yang et al. propose the PLCrypto library [51]. PLCrypto leverages the programming model of industrial Programmable Logic Controllers, where only data is remotely accessible. Hence, it stores cryptographic secrets in code to prevent manipulation.

8 Conclusion

This paper presented Lixom, a novel and generic technique for protecting cryptographic secrets with execute-only memory. Lixom-Light defends against transient execution attacks, and Lixom additionally protects secrets from the kernel of a VM guest. We implemented 3 case studies for Lixom, showing its applicability to password checking, AES encryption, and message authentication. Our performance studies showed that Lixom works with real-world software and can achieve a better throughput for AES than OpenSSL. Lixom can serve as a low-cost hardening technique in many applications using AES, and its unique properties are particularly well-suited for DRM systems.

Acknowledgment

We would like to thank our anonymous shepherd, and our anonymous reviewers for their insightful and constructive feedback.

References

1. "INTEL-SA-00898: 2024.1 IPU - Intel Atom Processor Advisory," 2024, accessed 2024-08-03. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00898.html
2. "AMD64 Architecture Programmer's Manual," Advanced Micro Devices Inc., 2024.
3. 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 *ACM SIGSAC CCS*, 2014.
4. M. Bauer, L. Hetterich, M. Schwarz, and C. Rossow, "Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8," in *AsiaCCS*, 2024.
5. P. Borrello, A. Kogler, M. Schwarzl, M. Lipp, D. Gruss, and M. Schwarz, "ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture," in *USENIX Security*, 2022.
6. K. Braden, L. Davi, C. Liebchen, A.-R. Sadeghi, S. Crane, M. Franz, and P. Larsen, "Leakage-resilient layout randomization for mobile devices." in *NDSS*, vol. 16, 2016.
7. S. Brookes, R. Denz, M. Osterloh, and S. Taylor, "Exoshim: Preventing memory disclosure using execute-only kernel code," *International Journal of Information and Computer Security*, 2022.
8. C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.
9. 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 *USENIX Security*, 2019.

10. D. Chakraborty, M. Schwarz, and S. Bugiel, “TALUS: Reinforcing TEE Confidentiality with Cryptographic Coprocessors,” in *FC*, 2023.
11. G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, “SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution,” in *EuroS&P*, 2019.
12. 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 *IEEE SP*, 2015.
13. B. Garmany and T. Müller, “Prime: private rsa infrastructure for memory-less encryption,” in *ACSAC*, 2013.
14. J. Gionta, W. Enck, and P. Larsen, “Preventing kernel code-reuse attacks through disclosure resistant code diversification,” in *IEEE CNS*, 2016.
15. L. Guan, J. Lin, B. Luo, and J. Jing, “Copker: Computing with private keys without ram.” in *NDSS*, 2014.
16. L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, “Protecting private keys against memory disclosure attacks using hardware transactional memory,” in *S&P*, 2015.
17. 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 *USENIX ATC*, 2019.
18. Intel, “Intel 64 and IA-32 Architectures Software Developer′s Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4,” 2024.
19. 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.
20. 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 *S&P*, 2019.
21. D. Kwon, J. Shin, G. Kim, B. Lee, Y. Cho, and Y. Paek, “uXOM: Efficient eXecute-Only memory on ARM Cortex-M,” in *USENIX Security*, 2019.
22. M. Larabel and M. Tippett, “Phoronix test suite,” *Phoronix Media*, 2011, accessed 2024-08-03. [Online]. Available: https://www.phoronix-test-suite.com/
23. 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*, 2021.
24. M. Li, Y. Zhang, and Z. Lin, “CrossLine: Breaking “Security-by-Crash” based Memory Isolation in AMD SEV,” in *SIGSAC*, 2021.
25. 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 *USENIX Security*, 2018.
26. G. Maisuradze and C. Rossow, “ret2spec: Speculative Execution Using Return Stack Buffers,” in *CCS*, 2018.
27. Microsoft, “Virtualization-based security (vbs),” 2021. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs
28. D. Moghimi, “Downfall: Exploiting speculative data gathering,” in *USENIX Security*, 2023.
29. M. Morbitzer, M. Huber, J. Horsch, and S. Wessel, “Severed: Subverting amd’s virtual machine encryption,” in *EuroSec*, 2018.
30. T. Müller, F. C. Freiling, and A. Dewald, “TRESOR Runs Encryption Securely Outside RAM,” in *USENIX Security*, 2011.
31. T. Ormandy, “Zenbleed,” 2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

32. D. A. Osvik, A. Shamir, and E. Tromer, “Cache Attacks and Countermeasures: the Case of AES,” in *CT-RSA*, 2006.
33. G. Patat, M. Sabt, and P.-A. Fouque, “Exploring widevine for fun and profit,” in *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2022.
34. 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 *EuroSys*, 2017.
35. M. Schink and J. Obermaier, “Taking a look into Execute-Only memory,” in *USENIX WOOT*, 2019.
36. M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, “ZombieLoad: Cross-Privilege-Boundary Data Sampling,” in *CCS*, 2019.
37. M. Seaborn and T. Dullien, “Exploiting the dram rowhammer bug to gain kernel privileges,” *Black Hat*, vol. 15, no. 71, 2015.
38. M.-W. Shih, S. Lee, T. Kim, and M. Peinado, “T-SGX: Eradicating controlled-channel attacks against enclave programs,” in *NDSS*, 2017.
39. S. Sparks and J. Butler, “Shadow walker: Raising the bar for rootkit detection,” *Black Hat Japan*, vol. 11, no. 63, 2005.
40. J. Torrey, “More shadow walker: Tlb-splitting on modern x86,” *Blackhat USA*, 2014.
41. D. Trujillo, J. Wikner, and K. Razavi, “Inception: Exposing new attack surfaces with training in transient execution,” in *USENIX Security*, 2023.
42. C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A practical library OS for unmodified applications on SGX,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.
43. P. Turner, “Retpoline: a software construct for preventing branch-target-injection,” 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886
44. 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 *USENIX Security*, 2019.
45. J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution,” in *USENIX Security*, 2018.
46. 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*, 2019.
47. A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert, “You shall not (by) pass! practical, secure, and fast pku-based sandboxing,” in *EuroSys*, 2022.
48. O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, “Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution,” 2018.
49. J. Wikner and K. Razavi, “Retbleed: Arbitrary speculative code execution with return instructions,” in *USENIX Security*, 2022.
50. 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, 2015.
51. Z. Yang, Z. Bao, C. Jin, Z. Liu, and J. Zhou, “Plcrypto: A symmetric cryptographic library for programmable logic controllers,” *IACR Transactions on Symmetric Cryptology*, 2021.
52. R. Zhang, L. Gerlach, D. Weber, L. Hetterich, Y. Lü, A. Kogler, and M. Schwarz, “CacheWarp: Software-based Fault Injection using Selective State Reset,” in *USENIX Security*, 2024.