PortPrint: Identifying Inaccessible Code with Port Contention

Tristan Hornetz and Michael Schwarz

CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
<firstname>.<lastname>@cispa.saarland

Abstract. In many real-world scenarios, being able to infer specific software versions or variations of cryptographic libraries is critical to mounting targeted exploits. For this, traditional version-detection approaches often rely on direct inspection of programs. However, modern computing platforms frequently employ protection for code, e.g., using execute-only memory (XOM) or trusted execution environments (TEE) to safeguard sensitive code from disclosure and reverse engineering.
This paper demonstrates how side-channel measurements via CPU port contention reveal distinctive execution signatures, even when code is inaccessible for inspection. Our proof-of-concept implementation PortPrint identifies cryptographic functions, reveals library versions, and even uncovers whether a WolfSSL build is vulnerable to CVE-2024-1544 or if Spectre mitigations are active in Xen. We verify that PortPrint works despite state-of-the-art code protection mechanisms, such as memory protection keys, hypervisor-based XOM, Intel SGX, Intel TDX, and AMD SEV. We also report a negative result for leaking code protected with these techniques using Meltdown and Foreshadow, providing valuable insights into the limitations of these attacks. Our results show that hardware-based isolation is insufficient to conceal instruction streams.

1 Introduction

In practical attack scenarios, knowing the exact software version or cryptographic library variant a victim uses is often crucial for mounting successful attacks. Thus, one strategy for improving system security is to prevent unauthorized code inspection. Execute-only memory (XOM) achieves this by allowing the CPU to fetch and run instructions without making them accessible to load or store instructions. Early implementations date back to 1965 [12], with several later works proposing XOM for security hardening, e.g., to protect intellectual property [31], or impede exploitation [5, 8, 14, 21, 30]. Consequently, CPU vendors also added hardware support for XOM, e.g., with Memory Protection Keys (MPK), which allow marking pages as execute-only [23, 33]. In parallel, Trusted Execution Environments (TEEs), such as Intel SGX, AMD SEV, or Intel TDX, rely on transparent memory encryption at the hardware level to prevent code inspection.

While XOM and TEEs offer strong security guarantees, previous work has shown that microarchitectural attacks can undermine these protections [10, 40]. Data inside TEEs has been a prime target for microarchitectural attacks, as TEEs have strong attackers in their threat model and–by definition–contain sensitive data [40]. Moreover, transient-execution attacks such as Meltdown [32], Foreshadow [46], RIDL [38], and ZombieLoad [41] demonstrate that microarchitectural attacks can leak data despite strong hardware protection. However, these attacks only focus on disclosing *data* instead of *code*.

In this paper, we demonstrate that port contention attacks [3] effectively leak information about the instructions executed in environments where the code is inaccessible, such as with XOM or TEEs. By exploiting simultaneous multithreading (SMT), attackers can measure timing delays that occur when different instruction types contend for specific execution ports. Since this method monitors the *instruction* pipeline rather than the *data* pipeline, it circumvents mechanisms such as memory encryption or XOM that primarily secure data paths. As a result, attackers can reveal high-level code characteristics, which suffices for fingerprinting cryptographic functions, library versions, and other unique code fragments without needing to read the instructions directly. Our proof-of-concept implementation PortPrint also works in scenarios where the code is protected by state-of-the-art XOM techniques [21, 33] or TEEs such as Intel SGX, Intel TDX, or AMD SEV. Our evaluation of all these techniques shows that they have no impact on the port-contention leakage.

We systematically evaluate different microarchitectures, including Intel (Kaby Lake, Golden Cove) and AMD (Zen 3). We evaluate the technique on four symmetric ciphers–AES, ChaCha20, Camellia, and Aria–as well as the SHA1, SHA256, SHA3, and MD5 hash algorithms and demonstrate that different implementations can be distinguished. Additionally, even the library (Linux vs. OpenSSL) can be reliably detected. We demonstrate that the subtle differences in port usage of instructions are enough to build a classifier with an accuracy above 99 % for cryptographic workloads. In a realistic scenario, we detect whether a version of WolfSSL is vulnerable to CVE-2024-1544 by observing port contention while it performs ECDSA signing. Furthermore, we show how an attacker can identify which Spectre mitigations are enabled in Xen by measuring the hypervisor's port usage on `cpuid` traps. Our approach informs attackers about possible exploits by revealing the cryptographic implementation and potential mitigations. For instance, detecting a vulnerable library version can help an attacker use a tailored microarchitectural attack.

Interestingly, while transient-execution attacks such as Meltdown [32], Foreshadow [46], and ZombieLoad [41] can leak data from microarchitectural buffers, our experiments confirm that they are ineffective in leaking *code*. This is a consequence of their reliance on data-path vulnerabilities and the lack of a direct equivalent in the instruction pipeline. In contrast, port contention exploits fundamental design features of SMT and execution ports, showing that hardware-based isolation through XOM or TEEs alone may be insufficient to conceal instruction streams.

Contributions. In summary, we make the following contributions:

- We present PortPrint, a generic port contention–based technique that reveals fine-grained instruction usage in protected code, including XOM or TEE settings.
- We demonstrate its effectiveness in three case studies: (1) fingerprinting cryptographic algorithms and their implementations (Linux vs. OpenSSL), (2) distinguishing between vulnerable and fixed WolfSSL implementations (CVE-2024-1544), and (3) detecting whether Spectre mitigations are active in Xen.
- We compare our approach with transient-execution vulnerabilities (Meltdown, Foreshadow, ZombieLoad). We observe that these attacks fail to leak code, underscoring that port contention is a unique threat vector against code confidentiality.

Outline. The remainder of the paper is organized as follows. Section 2 introduces the background required for the paper. Section 3 describes the design and implementation of PortPrint. Section 4 evaluates PortPrint on 3 case studies, showing that it can distinguish cryptographic implementations and library versions and detect the presence of Spectre mitigations in the hypervisor. Section 5 discusses the security impact, limitations, mitigations, applicability to other architectures, and related work. Section 6 concludes.

Availability. We publish proof-of-concept code for PortPrint under `https://github.com/cispa/PortPrint`.

2 Background

In this section, we provide the background required for the remainder of the paper. We discuss the state of the art for execute-only memory and introduce port contention.

2.1 Execute-Only Memory (XOM)

Execute-only memory prevents direct read access while allowing instruction fetches and execution. With the growing number of code-reuse exploits, researchers have advocated XOM to impede the disclosure of binaries [5, 7, 43]. For exploitation techniques like return-oriented programming to be feasible, an attacker must have access to the target binary's code. When denying read access to code sections, an attacker cannot easily learn gadget addresses, thus complicating return-oriented programming (ROP) or other code-reuse techniques. Several proposals also combine XOM with code diversification to achieve leakage resistance [14, 30, 42].

2.2 Hardware-enforced XOM

Protection Keys. On x86 CPUs, modern hardware-based XOM can be created using Memory Protection Keys (MPK) [1, 23], available on multiple Intel and

AMD CPUs. MPK reserves a 4-bit "protection key" in each page table entry, mapped to bits in a dedicated `pkru` register. User-mode software can update this register without kernel transitions to selectively disable read or write access to pages tagged with the same key. With both bits set, the code in those pages is effectively execute-only.

Since MPK aims to prevent accidental memory corruption rather than malicious attacks, some security weaknesses persist [11, 20, 45, 48]. For instance, file-backed memory remains mutable, and `sigreturn`-based manipulations can revert `pkru` settings. Still, MPK offers a low-overhead XOM mechanism.

XOM in Virtual Machines. The virtualization extensions in recent x86 CPUs give hypervisors fine-grained control over memory access permissions in virtual machines. With features such as Intel's Extended Page Tables (EPT) [23] and AMD's Reverse Map Table (RMP) [1], this also includes the capability to create XOM ranges in the guest's address space. Hence, these mechanisms can serve as a more secure alternative to MPK in virtualized environments [8, 14, 21].

Trusted Execution Environments and Confidential VMs. Hardware-backed trusted execution environments (TEEs) also implement memory encryption and isolation schemes that can effectively result in execute-only properties. Examples include Intel SGX, AMD SEV, and, more recently, Intel TDX. In particular, AMD Secure Encrypted Virtualization (SEV) [28] provides transparent memory encryption for entire virtual machines, ensuring that a malicious hypervisor or privileged adversary cannot read the contents of guest memory. Later iterations, such as SEV-ES and SEV-SNP, tighten the security guarantees by hiding CPU register states and adding integrity checks on guest memory, respectively. Intel Trust Domain Extensions (TDX) [22] similarly isolate tenants' confidential virtual machines (Trust Domains) from a potentially untrusted virtual machine manager or hypervisor. By encrypting guest pages and restricting hypervisor access to them, TDX prevents memory inspection or tampering.

2.3 Port Contention

Modern CPUs rely on simultaneous multithreading (SMT) to improve core utilization by enabling two or more logical threads to operate on shared physical resources. While these shared structures boost overall throughput, they also expose potential information leaks if one thread can observe the behavior of another through resource contention. When the CPU frontend decodes an instruction, the instruction is decomposed into one or more smaller micro-operations (μops). These μops are then dispatched to specific execution ports that map to specialized functional units. Each port is designed to handle certain classes of operations (e.g., arithmetic, load/store, and floating point). However, particular μops can sometimes be routed to multiple ports.

Because SMT allows multiple threads to run in parallel on the same core, contention arises if both threads compete for identical execution ports, resulting in timing variations. Attackers can exploit this by crafting *contention primitives*, which consist of instructions that only target a specific port. By analyzing their execution times, an attacker can thus infer how frequently the sibling thread

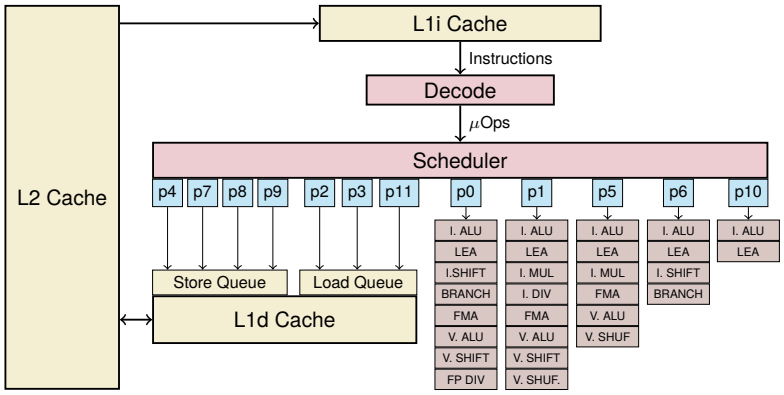


Fig. 1: Instruction processing in Intel's Golden Cove microarchitecture [26]. The scheduler dispatches μOps through the execution ports p0 - p11.

uses this port. Prior works demonstrate that this can leak sensitive information in cryptographic implementations [3] or be used for Spectre attacks [6]. Recent research has extended these attacks to the browser setting, revealing that side-channel adversaries can infer port usage via WebAssembly [37], effectively enabling cross-thread leakage in sandboxed environments.

For illustration, see Figure 1, which shows the Golden Cove microarchitecture used in the performance cores of Intel's Alder Lake CPUs. It has 12 ports (p0-p11) in total, 7 of which handle memory accesses and 5 of which handle branches and arithmetic. Due to extensive reverse-engineering efforts [2], port usage of instructions in this architecture is well known. To create a contention primitive, an attacker could e.g., use `crc32` instructions, which only target p1. However, ports like p6 cannot be probed individually, as every instruction executed through p6 also utilizes p0. Following the convention of Abel et al. [2], we denote contention primitives targeting such combinations with both port numbers, e.g., targeting p06.

3 Design and Implementation of PortPrint

This section presents the design of PortPrint. We provide an overview in Section 3.1, define the attacker model in Section 3.2, and explain our design and implementation of PortPrint in Section 3.3 and Section 3.4.

3.1 Overview

On a high level, PortPrint's objective is to differentiate between different configurations of the victim code. For instance, if the victim uses a cryptographic library, a potential goal is to find out the specific algorithm it uses or to check whether the implementation is prone to known vulnerabilities. PortPrint achieves

Fig. 2: PortPrint has four stages: (1) Creating port-contention primitives in the offline phase either dynamically or based on reverse-engineered information, (2) tracing the execution ports during victim execution, (3) converting the traces to a port-contention profile, and (4) classifying the profiles.

this by tracing the execution times for instructions targeting a specific execution port while the victim executes in the sibling thread, similar to attacks like PortSmash [4] and SMoTherSpectre [6]. It then combines multiple of these measurements into a *port contention profile*, which indicates how contended a specific port is n cycles after starting the victim code. We show that this port contention profile is specific to the victim code and can accurately identify a known program.

3.2 Attacker Model

PortPrint assumes an attacker capable of executing arbitrary code in the victim's SMT sibling thread. Furthermore, this attacker possesses a synchronization primitive in the victim's code that is accurate within roughly 100 cycles. Hence, they can temporally align repeated measurements for different ports. We assume that the execution of the victim code frequently repeats to facilitate this. This is in line with the threat model of TEEs, where a privileged attacker may use single-stepping for synchronization [47, 50, 51]. With attacks against the kernel, hypervisor, or XOM, an attacker can invoke the victim code directly, eliminating the need for synchronization.

3.3 Design

PortPrint consists of 4 stages as illustrated in Figure 2: (1) finding port contention primitives, (2) tracing the victim port usage, (3) converting measurements to profiles, and (4) classifying the profiles.

Port Contention Primitives. We refer to a block of instructions that, when executed, exhausts only a single execution port or a specific combination of execution ports as a *port contention primitive*. The port usage of instructions is undocumented and subject to change between microarchitectures, and these port contention primitives are hence specific to the target microarchitecture. PortPrint's contention primitives are based on the port usage information reverse-engineered by Abel et al. [2]. However, if this information is unavailable, it can also be obtained with dynamic approaches, as previous work has demonstrated [15, 18, 49].

Algorithm 1: Convert Time Intervals to Contention Profiles

Input : N: Number of desired profile samples.
δ: Number of cycles between profile samples.
T: Array of TSC intervals, covering at least $N \times \delta$ cycles
Output: P: Contention profile with N samples.

$p \leftarrow 0$;
$\mathsf{S} \leftarrow \text{CumulativeSum}(\mathsf{T})$;
for $i \leftarrow 0$ **to** $N - 1$ **do**
  while $\mathsf{S}[p] < i \times \delta$ **do**
    $p \leftarrow p + 1$;
  end
  $\mathsf{P}[i] \leftarrow \mathsf{T}[p]$;
end
return P;

Tracing the Victim's Port Usage. The attacker continuously executes a port contention primitive while the synchronized victim is executing. Once the primitive's execution is complete, the attacker records the current time-stamp counter value (i.e., the value of `rdtsc`) and repeats this process for a certain number of iterations. This leaves them with a trace of time stamps, which they can trivially convert into time intervals, and hence the execution times of the contention primitives. Larger intervals indicate a higher level of contention for the target port at the time of recording, meaning that the victim must have executed port-specific instructions. For better accuracy, PortPrint records multiple such traces for each execution port and repeats this procedure for multiple ports. Note that while we can precisely probe ports for arithmetic operations, this is significantly more challenging for ports that control memory accesses or branches since timing differences from branch prediction and caching structures can far outweigh the effects of port contention. Therefore, we only consider execution ports we can probe without accessing memory or inducing control-flow speculation.

Creating Port Contention Profiles. Although the attacker possesses the information required to differentiate between code configurations, performing this classification directly on time-stamp traces is difficult. This is primarily because the effects of port contention may only amount to a few cycles of additional latency between measurements, leading to a low signal-to-noise ratio in the measurements [18]. Furthermore, a single outlier can skew the timing of all following measurements.

PortPrint solves this problem by first converting the time-stamp traces to *port contention profiles*. Instead of latencies between measurements, such profiles record the time needed to execute the port contention primitive at a given time after starting the victim procedure. To perform this conversion, we use Algorithm 1, which takes a profile length N, a profile sample interval δ, and the

sequence of measured time stamp intervals T. In $\mathcal{O}(|T| + N)$ time, it creates a contention profile P of length N, where $P[i]$ is the interval observed after $i \times \delta$ cycles. Such profiles have the advantage of being temporally aligned. This allows us to combine multiple profiles from the same victim program into one by using element-wise arithmetic means, thus improving the signal-to-noise ratio.

Profile Classification. Finally, PortPrint uses a classifier model to determine the victim's code configuration. While many classification algorithms may work for this purpose, our PortPrint design uses simple Random Forest Classifiers with 100 decision trees for binary and multi-class classification. We train this classifier on profiles of known code configurations, allowing us to automatically distinguish between them.

3.4 Implementation

Our proof-of-concept implementation for PortPrint consists of two components: A *profiler*, which records the time-stamp traces, and a *classifier*, which converts them to port contention profiles and classifies them with a Random Forest classifier.

Profiler. Our profiler is implemented in C and small segments of assembly for the port contention primitives. The proof-of-concept implementation used in this paper targets the Intel Skylake, Intel Golden Cove, and AMD Zen 3 microarchitectures. We list the instructions used by our port contention profiles in Appendix B. While these might not be the ideal primitives, they work well for a wide range of scenarios, as shown in Section 4.

Classifier. The classifier is implemented in Python. It relies on the popular scikit-learn library [34], which provides a highly optimized Random Forest Classifier implementation.

4 Evaluation

We evaluate the effectiveness of PortPrint in 3 case studies: Section 4.1 demonstrates the general feasibility of our approach by distinguishing between different cryptographic algorithms based on port contention. In Section 4.2, we explore a more practical scenario by distinguishing between a vulnerable and a non-vulnerable version of the popular WolfSSL library. Finally, we use PortPrint to determine which Spectre mitigations are enabled in the Xen hypervisor in Section 4.3.

Evaluation Setting. We extensively evaluate PortPrint on an Intel Core i3 8130U (Kaby Lake), an Intel Core i5 13600KF (Alder Lake with Golden Cove P-Cores), and a Ryzen 7 5700G (Zen 3 Cezanne). Additionally, we verify PortPrint's applicability with various code protection mechanisms, including AMD's SEV on an AMD EPYC 7252, Intel's TDX on an Intel Xeon Gold 6526Y, and XOM (both MPK and hypervisor-enforced) on an Intel Core i5 13600KF . The details about the tested systems are listed in Table 1 in Appendix A. We observe the same results independent of the protection used, which is expected, as these protection methods do not interfere with execution units.

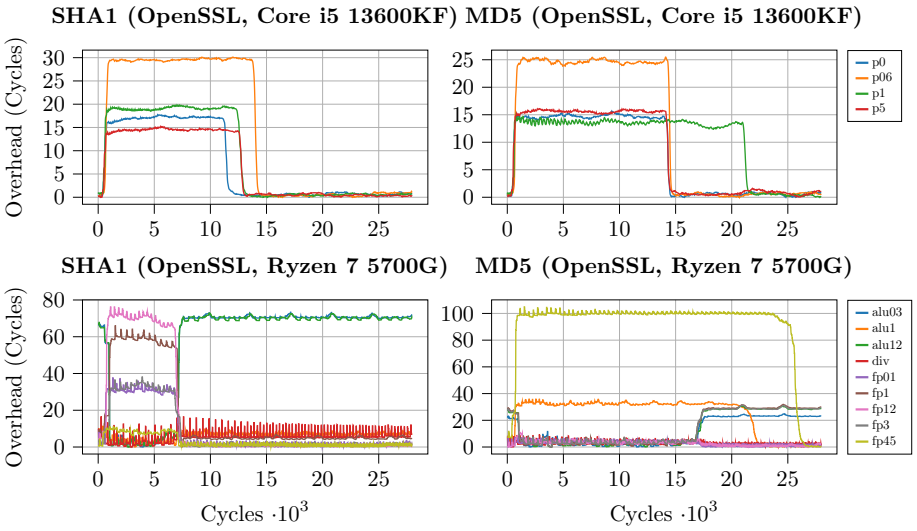


Fig. 3: Average runtime overhead of the port contention primitives when executing cryptographic algorithms in the sibling thread (average of 2^{15} samples).

4.1 Distinguishing Cryptographic Algorithms

We demonstrate that PortPrint can distinguish between cryptographic algorithms by profiling symmetric ciphers and hash algorithms from Linux and OpenSSL. Overall, we evaluate 17 implementations, as listed in Appendix C. We profile each implementation by invoking them with a workload size of 4 kB while continuously measuring the execution time of our contention primitives. These raw measurements are then converted into contention profiles with $N = 4000$ and $\delta = 20$ cycles, thus covering 80 000 cycles overall.

Hash Algorithms. Figure 3 shows excerpts from the contention profiles of OpenSSL's SHA1 and MD5 implementations. The two algorithms are easily distinguishable by their profiles. In contrast, the execution times are similar, making it difficult to distinguish them purely via timing. Note that the execution times can differ depending on which port we contend for in the sibling thread. For instance, MD5 takes roughly 50 % longer on the Core i5 13600KF if we contend for p1 in the sibling thread. Contrarily, SHA1 is not as drastically affected by this effect.

Symmetric Ciphers. We also apply PortPrint to common symmetric ciphers, again using side-by-side implementations from both Linux and OpenSSL. Although their respective cipher versions often share assembly-level optimizations, we find that they still show distinct port contention profiles. For illustrative purposes, Figure 4 shows profiles of several ciphers using 128 bit keys on a Kaby Lake i7-7700K processor.

From looking at the profiles in Figure 4, it is evident that the port usage for the Linux and OpenSSL versions of the same cipher are very similar. This

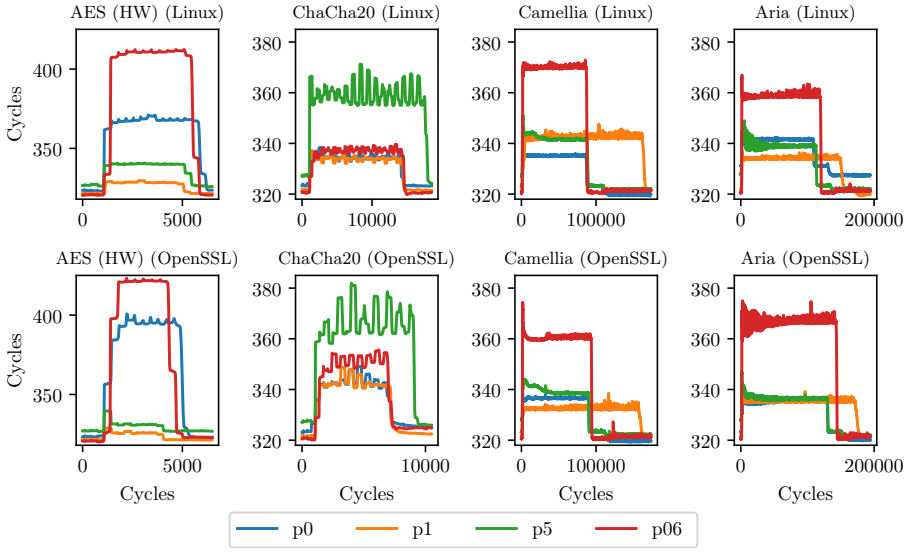


Fig. 4: Port contention timeline when encrypting 4 kB of data with various symmetric ciphers on an Intel Core i7-7700K (Kaby Lake). Each curve shows the average cost of probing a specific port at a given time ($n = 2048$).

is unsurprising, as both versions implement the same functionality and use the same instruction set extensions. Moreover, all ciphers except the used version of Aria are implemented in assembly, preventing the compiler from generating vastly different instructions.

Across AES, ChaCha20, Camellia, and Aria (all used in counter mode, except for ChaCha20, which is a stream cipher), different ports experience distinct slowdowns. Since AES implementations rely heavily on the `aesenc` instruction, which only uses p0 on Kaby Lake [2], that port is especially congested for AES. In contrast, ChaCha20 calls vector shuffle instructions, primarily executed on p5. Although Camellia and Aria appear more similar overall, p1 shows more significant contention with Camellia, almost doubling its encryption latency relative to Aria when this port contended. Moreover, when comparing Linux and OpenSSL implementations of the same cipher, the profiles remain nearly identical, likely due to the use of the same architecture-specific instructions and minimal compiler-driven changes.

Classification Performance. To automatically distinguish between all 17 implementations (including hash functions and ciphers), we record 1024 profiles per implementation and split them into training and test sets with a 25 % test set ratio. We train a multi-label Random Forest classifier on the training set and use the test set for cross-validation. This results in an average 1 vs. all F_1 score exceeding 99 % on all three CPUs, demonstrating that the port contention profiles can be easily distinguished, even for different implementations of the

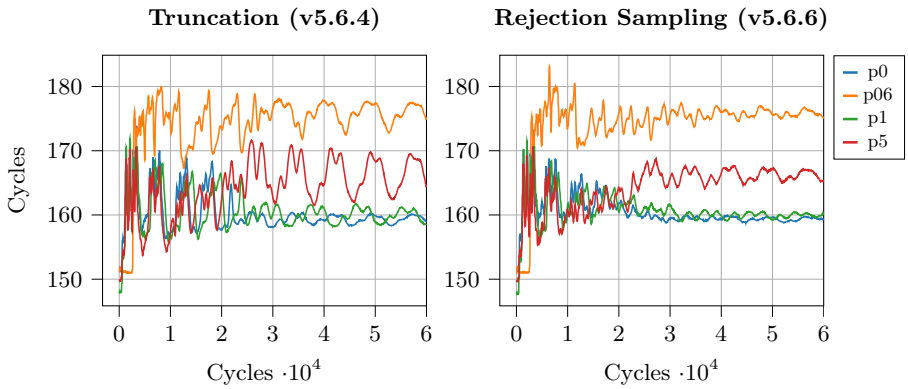


Fig. 5: Port contention profiles when signing a value with ECDSA-SECP160R1 for WolfSSL v5.6.4 with truncation-based nonce generation and v5.6.6 with rejection sampling (Intel Core i5 13600KF, Golden Cove). Only the former is affected by CVE-2024-1544.

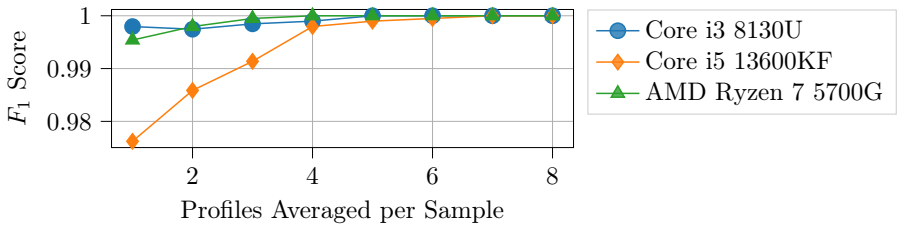


Fig. 6: F_1 score for classifying CVE-2024-1544 by number of profiles averaged in every sample.

same algorithm. Moreover, when repeating this test in a configuration where each of the 1024 samples is the average of 16 separate profiles, we achieve perfect accuracy on the Core i3 8130U and Core i5 13600KF. Only 3 test samples are falsely classified on the Ryzen 7 5700G, resulting in an F_1 score of 99.9 %. Hence, PortPrint can reliably identify cryptographic hash algorithms if instruction usage differs.

4.2 Spotting Vulnerable Implementations

This case study shows that PortPrint can distinguish between code versions prone to specific vulnerabilities and code versions where these vulnerabilities are fixed. For this purpose, we profile versions of WolfSSL with and without fixes for CVE-2024-1544. This vulnerability, discovered by Wilke et al. [50], allows a privileged attacker in a TEE setting to deduce bias in the most significant bits of the ECDSA nonce k. This can enable the attacker to fully recover the private key for certain curves. The root cause of this vulnerability is a control-flow

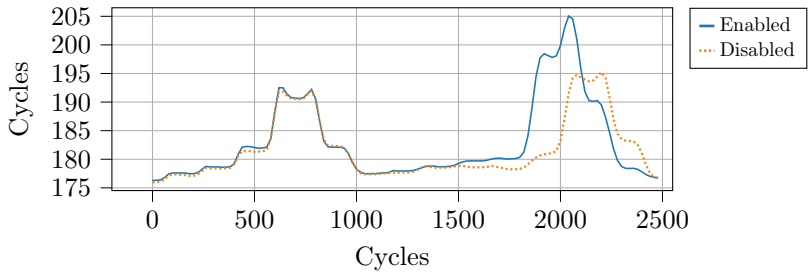


Fig. 7: Contention profile of p0 during a `cpuid` invocation on a Core i5 13000KF with and without BHI_DIS_S enabled in Xen (average of 2^{15} samples).

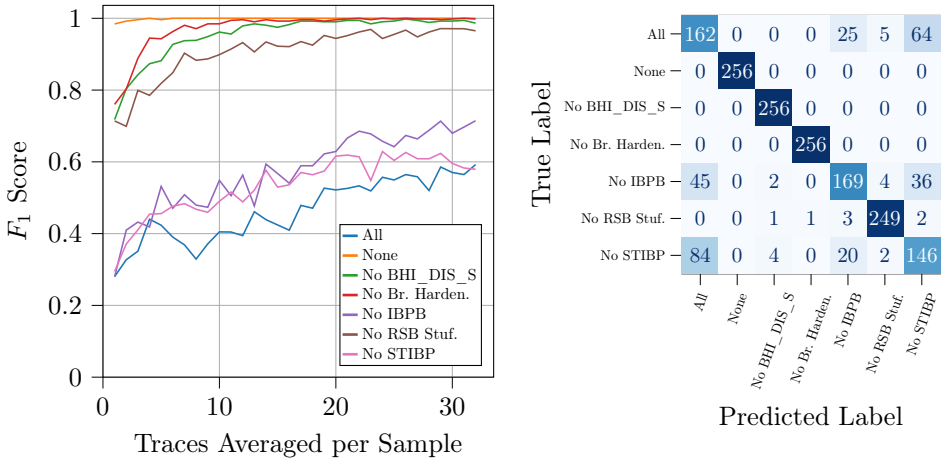
side channel in the modular reduction-based truncation algorithm that WolfSSL employs to generate k. To prevent this, WolfSSL also features an alternative implementation using rejection sampling in more recent versions, which does not allow this type of leakage. However, using rejection sampling over truncation is still optional and may be disabled in many real-world deployment scenarios.

We demonstrate that PortPrint can reliably determine whether a WolfSSL instance is prone to CVE-2024-1544 by observing ECDSA sign operations with curve SECP160R1. As with our first case study, we observe the execution times of our contention primitives and convert them into profiles with $N = 4000$ and $\delta = 20$ cycles. Averages of the resulting profiles for the Intel Core i5 13600KF are partially displayed in Figure 5. We then evaluate our classifier on 1024 profiles with a 25 % test set ratio. Figure 6 shows the resulting F_1 scores for the Core i3 8130U, the Core i5 13600KF, and the Ryzen 7 5700G. Again, the classifier's performance increases with the amount of profiles averaged into a sample. However, even with just one profile per sample, the worst F_1 score observed is 97.6 %. We reach perfect accuracy on all CPUs with 7 or more profiles per sample.

4.3 Identifying Spectre Mitigations

Finally, we explore PortPrint as a means to infer configuration details from higher privilege domains. Specifically, an attacker may be interested in whether specific Spectre mitigations are enabled before launching an attack. While kernels usually disclose this information without requiring privileges, this is not always the case for hypervisors. Thus, we perform this case study in a hardware-accelerated guest VM with Xen v4.19 on an Intel Core i5 13600KF CPU. To gain information about Spectre mitigations, we record a time stamp trace while invoking `cpuid` in the sibling thread. Since Xen enables trapping for `cpuid`, this causes a VM exit, allowing us to reliably observe a context switch from user mode to the hypervisor and, thus, the execution of Spectre mitigations.

On the Core i5 13600KF, Xen uses four Spectre mitigations that can be individually disabled: IBPB [24], STIBP [25], RSB Stuffing [13], and Intel's BHI_DIS_S predictor control setting [27]. Additionally, Xen hardens selected branches with speculation barriers, which can also be disabled at boot time. We



(a) 1 vs. all F_1 scores by number of averaged traces per sample.

(b) Confusion matrix with 32 averaged traces per sample.

Fig. 8: Classifier performance for detecting disabled Spectre mitigations (Core i5 13600KF / Golden Cove).

profile Xen in 7 configurations: one in which no mitigations are disabled, one in which all are disabled, and one in which each of the aforementioned countermeasures is individually disabled. Since the `cpuid` handler takes roughly 2500 cycles to execute, we convert the measured execution times into contention profiles with $N = 125$ and $\delta = 20$. For an example of a resulting contention profile, see Figure 7, which shows the profile for p0 with and without enabling the BHI_DIS_S control. Although these profiles are virtually identical initially, they start to diverge after roughly 1500 cycles, with p0 slightly less contended if BHI_DIS_S is disabled. This allows us to clearly distinguish between the two configurations, even if there is no timing difference in the `cpuid` handler's execution time.

We train our classifier like in the previous case studies, with 1024 profiles per configuration and a 25 % test set ratio. The results are shown in Figure 8. While configurations without IBPB and STIBP are hard to distinguish from configurations with all mitigations enabled, we achieve high accuracies for the remaining configurations. For configurations without any mitigations, we even achieve a perfect 1 vs. all F_1 score with 32 profiles per sample. Furthermore, even for configurations with more frequent misclassifications, we achieve accuracies far above 50 %. This demonstrates that there are still observable differences, although more subtle than with the other configurations. We theorize that more targeted profiling and classification methods can further improve accuracy.

4.4 Negative Result: Meltdown-type Attacks on Code

In addition to PortPrint, we evaluate whether transient-execution attacks have similar capabilities on affected CPUs. While these attacks are powerful in leak-

ing *data*, they have not been used for leaking *code*. We focus on Meltdown-type attacks, as they do not rely on specific gadgets in the victim that have to be exploited [29]. While Spectre-type attacks take advantage of mispredictions at branch boundaries to disclose unauthorized information, Meltdown-like techniques specifically target hardware protection checks to retrieve data the CPU should not make accessible [32]. Various attacks in this category exploit architectural or microarchitectural exceptions to leak from microarchitectural buffers [9, 35, 38, 41, 44]. These attacks allow an attacker to transiently access privileged content and exfiltrate it via covert channels like Flush+Reload.

We design experiments aiming to leak code on CPUs known to be susceptible to Meltdown-type attacks (e.g., Intel Core i7-7700K). We aim to align our experiments with the original Meltdown [32] and Foreshadow [46] experiments, i.e., we ensure that our targets are in the L1 cache. The main difference is that we target the L1i instead of the L1d. However, even with the targeted code in the L1i cache or executed on a sibling hyperthread, none of the existing Meltdown techniques are successful. Moreover, we do not see any instruction leakage from either hypervisor- or MPK-protected regions.

These observations are consistent with the absence of any documented mechanism that links the instruction-fetch pipeline to the buffers exploited by Meltdown-type attacks. Hence, attacks such as ZombieLoad [41], RIDL [38], or Fallout [10] cannot leak code, as the targeted buffers only contain data.

Still, data written from protected code into memory can become susceptible if it travels through Meltdown-vulnerable structures. In such scenarios, attackers could still infer some aspects of the code, such as stack operations or access to variables. However, the code itself cannot enter these structures during normal execution unless explicitly handled as data. Outside of JIT-compiled environments, this behavior is rare. With XOM, it cannot occur at all, as the hardware prevents data accesses to code.

5 Discussion

In this section, we discuss the security implications of PortPrint, its limitations, possible mitigations, its applicability to other architectures, and related work.

5.1 Security Implications

Our results demonstrate that leveraging port contention enables attackers to measure fine-grained instruction usage in execute-only environments. Consequently, XOM's promise of preventing unauthorized code inspection is weakened. The capacity to detect specific cryptographic routines or even software versions can inform attackers of potential vulnerabilities or known exploits. Moreover, the information gained from fingerprinting protected code may serve as a stepping stone for deeper analyses of proprietary algorithms or intellectual property, undermining the confidentiality that XOM aims to ensure.

Moreover, our attack methodology is broadly applicable. Since the leakage is inherent to the design of SMT, mitigating it is challenging. Therefore, the hardware-based isolation properties that XOM and specific Trusted Execution Environments rely upon are insufficient to guarantee that the sequence of executed instructions remains hidden.

Limitations. A fundamental requirement of our approach is the presence of SMT. If SMT is disabled or unavailable, PortPrint does not work. Moreover, the information leakage from port contention is limited to the granularity of the execution ports. While this level of detail can still reveal high-level instruction patterns (such as the usage of AES-NI or other specialized CPU instructions), reconstructing more specific aspects of the code might require further side channels or offline analysis. Thus, the approach is powerful for fingerprinting but does not provide a complete code dump.

Mitigations. A straightforward mitigation is to disable SMT. By ensuring that only one logical core shares execution resources at a time, we eliminate the source of port contention. This approach, however, has practical downsides. Disabling SMT can reduce overall system throughput, which might not be acceptable for high-performance computing environments.

Different Architectures. Our experimental results concentrate on x86 CPUs. However, our technique is not specific to x86. Any architecture with a comparable SMT design and similarly structured execution ports could be susceptible to port contention as used by PortPrint. Therefore, if other CPU vendors have similar shared-resource features for multithreading, PortPrint is likely applicable.

5.2 Related Work

This section discusses related work on port contention and breaking XOM.

Port Contention. Port contention as a microarchitectural side channel is not new and was initially analyzed by Aldaya et al. [3]. Similarly, SmotherSpectre [6] is an example of using this side channel as the covert channel in a Spectre attack, increasing the potential number of usable gadgets. Rokicki et al. [37] demonstrate port contention in the web browser context, achieving a cross-browser covert channel and demonstrating that port contention can remain robust against typical browser-based mitigations and noise. Covert Shotgun [15] highlights the possibility of automatically constructing covert channels based on microarchitectural contention. ABSynthe [18] presents an automated approach for detecting instruction sequences that create contention leaks and uses neural networks to extract cryptographic keys via carefully crafted instruction patterns.

Rokicki et al. [36] demonstrate that port contention can exist even without SMT, relying on instruction-level parallelism. Moreover, they analyze the portability of port contention across multiple Intel CPU generations, providing methods to detect suitable instruction sequences automatically.

SQUIP [16] reveals that contention in CPU scheduler queues can leak fine-grained information on modern microarchitectures, especially on machines that divide the scheduler among different execution units. SQUIP allows co-located

threads to extract cryptographic secrets. A follow-up paper shows that SQUIP can also be exploited via JavaScript and across different AMD CPUs [16].

Our work extends these existing attack strategies by specifically targeting execute-only memory and showing that even modern TEE or XOM configurations are not immune to port contention. While these earlier works focused on general data leakage, scheduler contention, or covert channel constructions, our emphasis on fingerprinting specific code segments within protected environments highlights the need for broader defenses. In particular, the applicability of port contention to settings where code is inaccessible, and the possibility to fingerprint cryptographic routines has not been explored to this extent.

Breaking Execute-only Memory. The concept of XOM first appeared in the Multics operating system on the GE 645 mainframe [12], where it was used to conceal sensitive code such as classroom grading programs. Early systems continued using XOM primarily for privacy preservation, but its scope later expanded. Yarvin et al. [52] showed how "anonymity" in a 64-bit address space could facilitate inter-domain communication while preventing address scans, motivating execute-only protections. Lie et al. [31] introduced a memory-encryption-based scheme to thwart code tampering and unauthorized redistribution, a precursor to modern trusted execution environments (TEEs) [19, 28].

Despite these protections, recent studies show that XOM can be undermined by advanced microarchitectural and speculation-based techniques. For instance, Schink et al. [39] highlight how firmware protected by microcontroller-level XOM remains vulnerable through shared CPU and SRAM resources, enabling full code recovery. In contrast to Schink et al. [39], we target modern x86 CPUs and not small microcontrollers. Göktas et al. [17] demonstrate a "speculative probing" approach that circumvents strict kernel-level defenses by exploiting speculation as a crash-free mechanism to leak code layout information. Our main difference to Göktas et al. [17] is that we do not rely on gadgets inside the victim.

6 Conclusion

We showed that modern protection mechanisms, including state-of-the-art XOM and TEEs (Intel SGX, Intel TDX, AMD SEV), cannot entirely prevent attackers from inferring which instructions a program executes. Our proof-of-concept implementation demonstrated that port contention enables accurate identification of cryptographic algorithms, library variants, and system configurations, even uncovering vulnerabilities such as CVE-2024-1544 or Spectre mitigations in Xen. Our negative result regarding Meltdown and Foreshadow further highlights that these transient execution attacks can only leak *data* rather than protected instructions.

Acknowledgment

We thank our anonymous reviewers for their insightful feedback. Additionally, we thank Lukas Gerlach for fruitful discussions contributing to this work.

References

1. AMD64 Architecture Programmer's Manual (2024)
2. Abel, A., Reineke, J.: uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In: ASPLOS (2019)
3. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port Contention for Fun and Profit. In: S&P (2018)
4. Aldaya, A.C., Brumley, B.B., ul Hassan, S., García, C.P., Tuveri, N.: Port contention for fun and profit. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE (2019)
5. Backes, M., Holz, T., Kollenda, B., Koppe, P., Nürnberger, S., Pewny, J.: You can run but you can't read: Preventing disclosure exploits in executable code. In: ACM SIGSAC CCS (2014)
6. Bhattacharyya, A., Sandulescu, A., Neugschwandtner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. In: CCS (2019)
7. Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., Boneh, D.: Hacking blind. In: S&P (2014)
8. Brookes, S., Denz, R., Osterloh, M., Taylor, S.: Exoshim: Preventing memory disclosure using execute-only kernel code. International Journal of Information and Computer Security (2022)
9. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking Data on Meltdown-resistant CPUs. In: CCS (2019)
10. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security (2019)
11. Connor, E., McDaniel, T., Smith, J.M., Schuchard, M.: PKU pitfalls: Attacks on PKU-based memory isolation systems. In: USENIX Security (2020)
12. Corbató, F.J., Vyssotsky, V.A.: Introduction and overview of the multics system. In: Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I (1965)
13. Corbet, J.: Stuffing the return stack buffer. LWN.net (2022)
14. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: Practical code randomization resilient to memory disclosure. In: IEEE SP (2015)
15. Fogh, A.: Covert Shotgun: automatically finding SMT covert channels (2016), https://cyber.wtf/2016/09/27/covert-shotgun/
16. Gast, S., Juffinger, J., Maar, L., Royer, C., Kogler, A., Gruss, D.: Remote scheduler contention attacks. In: Financial Cryptography and Data Security (2024)
17. Göktaş, E., Razavi, K., Portokalidis, G., Bos, H., Giuffrida, C.: Speculative Probing: Hacking Blind in the Spectre Era. In: CCS (2020)
18. Gras, B., Giuffrida, C., Kurth, M., Bos, H., Razavi, K.: ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In: NDSS (2020)
19. Gueron, S.: Memory encryption for general-purpose processors. IEEE SP (2016)
20. Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M.L., Shen, K., Marty, M.: Hodor:Intra-Process isolation for High-Throughput data plane libraries. In: USENIX ATC (2019)

21. Hornetz, T., Gerlach, L., Schwarz, M.: Lixom: Protecting Encryption Keys with Execute-Only Memory. In: FC (2025)
22. Intel: Intel Trust Domain CPU Architectural Extensions (2021)
23. 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)
24. Intel Corporation: Indirect Branch Predictor Barrier (2018)
25. Intel Corporation: Single Thread Indirect Branch Predictors (2018)
26. Intel Corporation: Intel 64 and IA-32 Architectures Optimization Reference Manual Volume 1 (2023)
27. Intel Corporation: Branch History Injection and Intra-mode Branch Target Injection (2024)
28. Kaplan, D., Powell, J., Woller, T.: AMD Memory Encryption (2016)
29. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
30. Kwon, D., Shin, J., Kim, G., Lee, B., Cho, Y., Paek, Y.: uXOM: Efficient eXecute-Only memory on ARM Cortex-M. In: USENIX Security (2019)
31. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. Acm Sigplan Notices (2000)
32. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading Kernel Memory from User Space. In: USENIX Security (2018)
33. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software Abstraction for Intel Memory Protection Keys. arXiv:1811.07276 (2018)
34. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research (2011)
35. Ragab, H., Milburn, A., Razavi, K., Bos, H., Giuffrida, C.: CrossTalk: Speculative Data Leaks Across Cores Are Real. In: S&P (2021)
36. Rokicki, T., Maurice, C., Botvinnik, M., Oren, Y.: Port contention goes portable: Port contention side channels in web browsers. In: AsiaCCS (2022)
37. Rokicki, T., Maurice, C., Schwarz, M.: CPU Port Contention Without SMT. In: ESORICS (2022)
38. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue In-flight Data Load. In: S&P (2019)
39. Schink, M., Obermaier, J.: Taking a look into Execute-Only memory. In: USENIX WOOT (2019)
40. Schwarz, M., Gruss, D.: How Trusted Execution Environments Fuel Research on Microarchitectural Attacks. IEEE Security & Privacy (2020)
41. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: ZombieLoad: Cross-Privilege-Boundary Data Sampling. In: CCS (2019)
42. Shen, Z., Dharsee, K., Criswell, J.: Fast execute-only memory for embedded systems. In: SecDev. IEEE (2020)
43. Snow, K.Z., Monrose, F., Davi, L., Dmitrien ko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: S&P (2013)
44. Stecklina, J., Prescher, T.: LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. arXiv:1806.07480 (2018)

45. Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Druschel, P.: ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. In: USENIX Security Symposium (2019)
46. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security (2018)
47. Van Bulck, J., Piessens, F., Strackx, R.: Sgx-step: A practical attack framework for precise enclave execution control. In: SysTEX (2017)
48. Voulimeneas, A., Vinck, J., Mechelinck, R., Volckaert, S.: You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In: EuroSys (2022)
49. Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., Rossow, C.: Osiris: Automated Discovery of Microarchitectural Side Channels. In: USENIX Security (2021)
50. Wilke, L., Sieck, F., Eisenbarth, T.: TDXdown: Single-Stepping and Instruction Counting Attacks against Intel TDX. In: CCS (2024)
51. Wilke, L., Wichelmann, J., Rabich, A., Eisenbarth, T.: Sev-step: A single-stepping framework for amd-sev. IACR Trans. Cryptogr. Hardw. Embed. Syst. (2024)
52. Yarvin, C., Bukowski, R., Anderson, T.: Anonymous rpc: Low-latency protection in a 64-bit address space. In: USENIX Summer (1993)

A Evaluation Setups

Table 1 lists the setups we use for evaluation, including the CPU, microcode version, and operating system.

B Contention Primitives

Our contention primitives use the instructions listed in Table 2. Note that the contention primitives for Skylake are also applicable to later Skylake-based microarchitectures, such as Kaby Lake [2]. Similarly, the primitives for Golden Cove apply to microarchitectures like Alder Lake and Sapphire Rapids [26].

CPU	Microcode	Operating System
Core i3 8130U	`0xf6`	Ubuntu 22.04 LTS w. Linux v5.15.0
Core i5 13600KF	`0x12b`	Debian 12 w. Linux v6.1.0
Xeon Gold 6526Y	`0x21000283`	Ubuntu 24.04 LTS w. Linux v6.8.0
Ryzen 7 5700G	`0xa50000d`	Ubuntu 22.04 LTS w. Linux v5.15.0
EPYC 7252	`0x830107c`	Ubuntu 22.04 LTS w. Linux v5.15.0

Table 1: Overview of used CPUs with microcode and operating system.

Port(s)	Skylake	Golden Cove
p0	`aesenc xmm, xmm`	`vrcpps xmm, xmm`
p06	`ror r64, i8`	`ror r64, imm8`
p1	`crc32 r64, r64`	`crc32 r64, r64`
p5	`vpermd ymm, ymm, ymm`	`vextractf128 xmm, ymm, i8`
Port(s)	**Zen 3**	
div	`idiv r8l`	
alu03	`cmovz r64, r64`	
alu1	`imul r64, r64`	
alu12	`ror r64, i8`	
fp01	`vbroadcastsd ymm, xmm`	
fp1	`vpinsrq xmm, xmm, r64, i8`	
fp12	`vpshuflw xmm, xmm, i8`	
fp45	`vmovmskps r64, ymm`	

Table 2: Instruction ports for Skylake, Golden Cove, and Zen3.

C Crypotographic Alogorithms

See Table 3 for the implementations of cryptographic algorithms used in Section 4.1. We compile implementations written in C with support for SSE3 using GCC v12.2.0 and optimization level `-O3`. For ciphers, we only invoke the encryption functions.

Algorithm	Source	Language	Extensions
AES-128-CTR (HW)	OpenSSL v3.2.1	ASM	AES-NI
AES-128-CTR (HW)	Linux v6.6.8	ASM	AES-NI
AES-128-CTR (SW)	OpenSSL v3.2.1	C	N.A.
Aria-128-CTR	OpenSSL v3.2.1	C	N.A.
Aria-128-CTR	Linux v6.6.8	C	N.A.
Camellia-128-CTR	OpenSSL v3.2.1	ASM	x64
Camellia-128-CTR	Linux v6.6.8	ASM	x64
ChaCha20	OpenSSL v3.2.1	ASM	SSE3
ChaCha20	Linux v6.6.8	ASM	SSE3
MD5	OpenSSL v3.2.1	ASM	x64
MD5	Linux v6.6.8	C	N.A.
SHA1	OpenSSL v3.2.1	ASM	SHA-NI
SHA1	Linux v6.6.8	ASM	SHA-NI
SHA256	OpenSSL v3.2.1	ASM	SHA-NI
SHA256	Linux v6.6.8	ASM	SHA-NI
SHA3-256	OpenSSL v3.2.1	C	N.A.
SHA3-256	Linux v6.6.8	C	N.A.

Table 3: Implementations used for the case study in Section 4.1.