ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture

Pietro Borrello[1], Andreas Kogler[2], Martin Schwarzl[2], Moritz Lipp[3], Daniel Gruss[2], Michael Schwarz[4]
[1] *Sapienza University of Rome,* [2] *Graz University of Technology,*
[3] *Amazon Web Services,* [4] *CISPA Helmholtz Center for Information Security*

Abstract

CPU vulnerabilities undermine the security guarantees provided by software- and hardware-security improvements. While the discovery of transient-execution attacks increased the interest in CPU vulnerabilities on a microarchitectural level, architectural CPU vulnerabilities are still understudied.

In this paper, we systematically analyze existing CPU vulnerabilities showing that CPUs suffer from vulnerabilities whose root causes match with those in complex software. We show that transient-execution attacks and architectural vulnerabilities often arise from the same type of bug and identify the blank spots. Investigating the blank spots, we focus on architecturally improperly initialized data locations.

We discover ÆPIC Leak, the first architectural CPU bug that leaks stale data from the microarchitecture without using a side channel. ÆPIC Leak works on all recent Sunny-Cove-based Intel CPUs (*i.e.*, Ice Lake and Alder Lake). It architecturally leaks stale data incorrectly returned by reading undefined APIC-register ranges. ÆPIC Leak samples data transferred between the L2 and last-level cache, including SGX enclave data, from the superqueue. We target data in use, e.g., register values and memory loads, as well as data at rest, e.g., SGX-enclave data pages. Our end-to-end attack extracts AES-NI, RSA, and even the Intel SGX attestation keys from enclaves within a few seconds. We discuss mitigations and conclude that the only short-term mitigations for ÆPIC Leak are to disable APIC MMIO or not rely on SGX.

1 Introduction

In recent years, a lot of research has been conducted to improve software security, both on the application layer as well as on the operating-system (OS) layer [35, 68]. The types of software vulnerabilities are well known and, e.g., categorized with the Common Weakness Enumeration (CWE) [70]. In addition to manual security analysis, there are several techniques to discover software vulnerabilities in automated and semi-automated ways, e.g., fuzzing [16, 85], or static and dynamic analysis [9, 58]. However, more recent works have shown that next to software vulnerabilities, there are software-exploitable hardware vulnerabilities, such as Meltdown [48] or Spectre [39]. These vulnerabilities can undermine software security which always assumes bug-free and secure hardware. The discovery of transient-execution attacks [4, 39, 48] showed that CPUs by virtually all vendors, including Intel, AMD, and ARM, are affected by these software-exploitable hardware vulnerabilities. However, as these vulnerabilities are architecturally not visible, transient-execution attacks use side channels to observe and exploit them architecturally.

Following Meltdown and Spectre, a multitude of transient-execution attacks has been discovered in this class of vulnerabilities [4, 38, 41, 63, 72, 78, 83]. All of these attacks leak data, Meltdown-type attacks even across security boundaries, including trusted execution environments (TEEs). Hence, they pose a severe threat to the system security and resulted in numerous ad-hoc mitigations on the operating-system and firmware level [3, 24]. Despite the significant amount of research on transient-execution attacks, they are not the only CPU vulnerabilities. Architectural bugs have been known for much longer, with infamous examples such as the Pentium FDIV bug [5] or the Pentium F00F bug [6]. These vulnerabilities are intuitively easier to observe as they do not require additional side channels. However, recent work has highlighted the difficulty of adequately testing CPU design for such vulnerabilities [11].

In this paper, we systematically analyze both architectural and transient-execution vulnerabilities, showing that the underlying type of vulnerability is often the same. While the CWE recently introduced categories for such hardware vulnerabilities, we show that the root cause of hardware vulnerabilities can also be classified using the existing vulnerability types for software. As CPUs are also written in (hardware) programming languages, it is indeed not surprising that vulnerabilities known from software are also present in hardware. However, we mainly see complex vulnerabilities in the hardware, such as race conditions or use after free.

Based on our systematic analysis, we investigate categories in which transient-execution attacks are known, but no archi-

tectural equivalent is known. Specifically, we systematically inspect CPUs for improperly initialized storage locations that return (parts of) stale data. We focus on data loads where the structure that holds the data is larger than the effective loaded data. Not initializing the whole structure may leave stale data in the region not overridden by the effective data. This is, e.g., the case in the I/O address space, where memory-mapped devices often have strict limitations, such as only allowing aligned 32-bit loads to specific addresses [28].

Discovering architectural leaks. The scan of the I/O address space on Intel CPUs based on the Sunny Cove microarchitecture revealed that the memory-mapped registers of the local Advanced Programmable Interrupt Controller (APIC) are not properly initialized. As a result, architecturally reading these registers returns stale data from the microarchitecture. Any data transferred between the L2 and the last-level cache can be read via these registers. This vulnerability, named ÆPIC Leak, affects the 10$^{\text{th}}$ generation mobile Ice Lake CPUs, the newest, 12$^{\text{th}}$ generation, Alder Lake CPUs, and the current 3$^{\text{rd}}$ generation of Xeon scalable server CPUs (Ice Lake SP).

As the I/O address space is only accessible to privileged software, ÆPIC Leak targets Intel's TEE, SGX. ÆPIC Leak can leak data from SGX enclaves that run on the same physical core. While ÆPIC Leak would represent an immense threat in virtualized environments, hypervisors typically do not expose the local APIC registers to virtual machines, eliminating the threat in cloud-based scenarios. Similar to previous transient-execution attacks targeting SGX [63, 72, 77–79], ÆPIC Leak is most effective when running in parallel to the enclave on the sibling hyperthread. However, ÆPIC Leak does not require hyperthreading and can also leak enclave data if hyperthreading is unavailable or disabled.

We present two new techniques to leak *data in use*, *i.e.*, values from enclave registers, and *data at rest*, *i.e.*, data stored in enclave memory. With Cache Line Freezing, we introduce a technique putting targeted pressure on the cache hierarchy without overwriting stale data. Cache Line Freezing exploits the observation that Sunny Cove implements an optimization for zero cache lines, *i.e.*, cache lines filled only with '0's. These cache lines still appear to travel through the cache hierarchy, but they do not overwrite stale data. With this targeted pressure and enclave single-stepping [74], we leak register values from cache lines in the secure state area (SSA). A second technique, Enclave Shaking, exploits the capability of the operating system to securely swap enclave pages. By alternatingly swapping enclave pages out and back in, the stored data is forced through the cache hierarchy, allowing ÆPIC Leak to leak the values without even continuing the execution of the enclave. We exploit ÆPIC Leak in combination with Cache Line Freezing and Enclave Shaking to extract AES-NI keys and RSA keys from Intel's IPP library and the Intel SGX sealing and remote attestation keys. Our attack leaks memory from enclaves with 334.8 B/s and a success rate of 92.2 %.

Although we provide software workarounds for specific scenarios, such as AES-NI, we conclude that there is no short-term workaround for protecting enclave data without disabling the APIC memory-mapped range or disabling SGX. On January 2022, Intel announced the deprecation of SGX on the affected CPU generations [32] for client architectures, which coincidentally reduces the risk of widespread exploitation after our submission. However, while it is deprecated on client CPUs, SGX is still available on server CPUs (*i.e.*, 3$^{\text{rd}}$ generation of Xeon scalable server CPUs). An attacker only needs one up-to-date system to extract secrets from an enclave (e.g., bypassing Signal private contact discovery [51], leaking DRM secrets or attestation keys). Thus, if not mitigated, exploiting ÆPIC Leak is a significant threat to enclave security. Disabling the APIC I/O memory via a microcode update, or deprecating SGX are effective mitigations against the specific vulnerability discovered. However, we argue that a generic mitigation of the vulnerability class in future hardware is an open research problem we identify with this work.

Contributions. The contributions of this work are:

1. We systematically analyze and categorize CPU vulnerabilities, showing that they have the same types as for software, and identifying blank spots.
2. In the blank spots, we discover ÆPIC Leak, an architectural vulnerability in the local APIC leaking data from SGX enclaves, including data in use and data at rest.
3. We design two complementary techniques that leverage microarchitectural optimizations to control which cache line ÆPIC Leak samples from the cache hierarchy.
4. We evaluate our techniques by leaking cryptographic keys, including Intel's official key from the quoting enclave.

Outline. Section 2 provides background. Section 3 analyzes and categorizes CPU vulnerabilities, leading to blank spots with potentially undiscovered vulnerabilities. Section 4 details the ÆPIC Leak vulnerability and its threat model. In Section 5, we show how ÆPIC Leak leaks data from SGX enclaves. We discuss mitigations in Section 6 and conclude in Section 7.

Responsible Disclosure. We responsibly disclosed our findings to Intel on December 8th, 2021. Intel acknowledged our findings on December 22nd, 2021, assigned CVE-2022-21233 and is working on possible mitigations.

Code Access. Our proof of concept of the attacks is open-sourced at `https://github.com/IAIK/AEPIC`.

2 Background

This section covers fundamental background for the reader to understand the rest of the paper.

2.1 APIC

The Advanced Programmable Interrupt Controller (APIC) manages and routes interrupts in modern CPUs. The APIC is split into two different components: The *Local APIC* integrated into each logical core and the external *I/O APIC*

in the Intel's System Chip Set. The Local APIC manages interprocessor interrupts (IPIs) and receives interrupts from the processor interrupts pins, forwarding them to the core to be handled, while the I/O APIC receives external interrupts events and forwards them to the target local APICs [28].

Local APIC. A Local APIC can receive, generate, and forward interrupts, both to its local core (through IPIs or local interrupt sources, *i.e.*, timer interrupts, performance monitoring counter interrupts, thermal sensor interrupts), to other cores (through IPIs), and from external devices (through the I/O APIC). Each Local APIC is made up of a set of APIC registers to control its functionality or expose the state of the interrupts in the system. A processor can generate IPIs or set up local interrupts via APIC registers of its own Local APIC.

Local APIC Registers. By default, modern APICs operate in xAPIC mode, which exposes Local APIC registers as a memory-mapped 4 kB region in the physical address space. The address of the region is set in the `IA32_APIC_BASE` MSR and independent for each logical core [28]. At startup, the region is set at physical address `0xFEE00000` but it can be moved on a per-core basis by changing the value of the `IA32_APIC_BASE` MSR. APIC registers are either 32, 64, or 256 bits, but they are mapped into the memory-mapped region as 32-bit values, always aligned to 128-bit boundaries. Thus, registers wider than 32 bits are split and mapped over multiple 128-bit aligned regions in the memory-mapped area. This means that bytes 4 to 15 in each 16-byte (128-bits) region are never architecturally defined. Intel states that *any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed* [28]. The APIC can be set in *x2APIC* mode if supported, which extends xAPIC mode with different improvements, like enhancing the performance of interrupt delivery and providing MSR-based access to APIC registers which disables the memory-mapped interface. The OS can enable or disable x2APIC mode by setting bit 10 of `IA32_APIC_BASE` MSR.

2.2 Memory Subsystem

CPUs rely on a hierarchical memory subsystem with data cached over multiple levels. Lower level caches provide faster memory with smaller storage capabilities for data frequently accessed, while higher-level caches offer bigger storage at cost of increased latency. Modern Intel CPUs usually have at the lowest level a private instruction cache (L1I) and data cache (L1D), and at the second level a private unified cache (L2). The Last-Level Cache (LLC or L3) is usually shared across all physical cores.

Path to Main Memory. The CPU tries to serve each memory access from the lowest cache level possible. It allocates the resources necessary to track the memory requests, *i.e.*, *load* or *store buffers*. Upon completion of the address translation, if any of the physical tags in the indexed cache set matches the physical address, the data is returned from the L1D. In case no tag matches (*i.e.*, the data is not in L1D), the CPU allocates a line-fill-buffer (LFB) entry to interface with the L2 cache. Line fill buffers act as a decoupling component between L1 and L2 caches to keep track of outstanding requests, uncacheable memory accesses, and non-temporal moves [63, 78]. The CPU then performs the lookup in L2, which loads the LFB entry in case the data is present, returns it to L1D, and back to the load buffers. In case data is not present in L2, the CPU must issue an offcore request to the LLC cache. It reserves a *fill buffer* entry to hold the data in the *superqueue* [40, 43] between L2 and LLC, issues the load over the *ring interconnect* [59] and waits for the request to be completed. The superqueue decouples the interaction between the L2 and the LLC caches, in a similar way the line fill buffers do between L1 and L2. The ring interconnect is an on-die interconnect used for uncore communication between CPU cores, LLC, memory controller, and the integrated GPU. The load request is satisfied either by the LLC or the memory controller, and the fill-buffer entry in the superqueue collects the value, which sends the data back to the core.

2.3 Intel SGX

Intel Software Guard Extension (SGX) provides a Trusted Execution Environment (TEE) on x86 processors. Introduced in Skylake CPUs, SGX offers hardware isolation and local and remote attestation for so-called *enclaves* even on possibly attacker-controlled machines [28]. SGX enclaves reside in the virtual address space of a userspace process, but their physical memory is backed by the protected Enclave Page Cache (EPC). Stores to EPC are automatically encrypted, and loads are decrypted by the memory encryption engine. While enclave memory is inaccessible to attackers probing the memory bus [7], CPUs affected by transient-execution vulnerabilities can leak the values from the microarchitecture [63, 72, 78].

Enclaves can only be executed from pre-configured entry points using the `eenter` instruction and exit using an `eexit` instruction. If a fault or interrupt occurs while an enclave is running, the processor issues an Asynchronous Enclave Exit (AEX), securely storing and clearing all the enclave CPU registers at the time of enclave interruption in a Save State Area (SSA) inside EPC. An `eresume` instruction restores enclave execution from the SSA frame.

Due to the limited EPC size, untrusted system software can leverage the `ewb` and `eldu` instructions to move encrypted EPC pages to main memory and back, without revealing the content. When an enclave page is moved from main memory back to EPC using `eldu`, it is decrypted and cryptographically verified to ensure its content has not been tampered with, bringing the plaintext data to the L1 cache [72].

SGX supports local and remote attestation. During the enclave creation process, the CPU collects cryptographic measurements about the starting enclave and its signature in two different Measurement Registers (`MRSIGNER` and `MRENCLAVE`). An enclave can generate a signed local attestation for a target enclave using the `ereport` instruction, which can be cryp-

tographically verified by the target enclave using a key obtained through the `egetkey` instruction. The report of the local attestation includes the enclave's initial code and data as measurement registers in addition to other security-related information [28]. Intel provides a trusted `quoting enclave` to sign locally-generated identity reports using an Intel-private key and enabling remote attestation. The `egetkey` instruction also provides a sealing key that the enclaves can use to securely `seal` secrets for untrusted persistent storage.

SGX enclaves have been compromised in numerous ways over the past years, e.g., memory-safety violations [44], insecure synchronization [82], asynchronous exception management [10], and side channels [53, 65, 75, 76]. SGX has also been the target of transient-execution attacks [62, 63, 72, 78].

2.4 Transient-Execution Attacks

On x86, the instruction stream, once fetched, is decoded into smaller micro-operations (μops) to simplify the underlying microarchitecture and enable low-level optimizations. The μops are decoded *in-order* and executed *out of order* over the different execution units, keeping track of the dependencies to satisfy them. The results are committed in order to the architecture, thus ensuring correctness. Given the highly parallel nature of modern CPUs, *branch prediction* has been introduced to avoid stalls, *speculatively* executing the predicted path. If the prediction turns out correct, the speculatively executed μops are committed to the architectural state, while in case of a misprediction, the results are discarded by the CPU. All non-committed μops are discarded if exceptions arise during out-of-order execution. Any discarded μop does not affect the architectural state, but it can affect the microarchitectural state (e.g., cache state). Such instructions are called transient instructions [4, 39, 48].

3 Software and Hardware Vulnerabilities

In this section, we systematically analyze existing documented CPU vulnerabilities on x86 CPUs, showing that the underlying root causes are the same as for software vulnerabilities. We demonstrate that the CWE classification of software vulnerabilities can be applied both to transient-execution vulnerabilities as well as architectural CPU vulnerabilities. Table 1 provides this classification.

3.1 Types of Vulnerabilities

For a long time, system security relied on the correctness of the underlying hardware, ignoring the possibility of security vulnerabilities on the CPU [37]. With the discovery of transient-execution attacks [39, 48], this view has changed drastically. Since the first publication of such attacks, numerous vulnerabilities have been discovered in CPUs [2, 4, 61–63, 72, 73, 78]. However, transient-execution attacks are neither the only nor the first discovered CPU vulnerabilities. The history of CPU vulnerabilities that affect a large number of users goes back to well-known bugs such as the Intel Pentium FDIV bug [5] described in 1995 or the Intel F00F bug [6] described in 1997. These bugs did not pose a significant security risk back then. However, today, with cloud computing and trusted-execution environments, such small bugs would be exploitable. DVFS attacks [36, 57, 60] can induce a similar effect as the FDIV bug by causing wrong results in multiplications (instead of divisions), which has been used to break the confidentiality and integrity of Intel SGX. Similarly, LVI-FP [61] induced wrong floating-point calculations in the transient domain, which has been exploited to disclose arbitrary memory in the browser. Hence, as we have seen with software, simple bugs can become exploitable vulnerabilities when exploitation techniques improve [20, 67].

When analyzing existing CPU vulnerabilities, we can–at the high level–categorize them into *architectural* and *transient vulnerabilities*. Architectural vulnerabilities are exploitable by relying only on architecturally-defined interfaces and features. Transient vulnerabilities do not have an architecturally-visible effect as they are only visible on the microarchitectural level and hence require side channels to observe them.

Architectural Vulnerabilities. Architectural vulnerabilities are visible without requiring any further indirection or side effects. x86 CPUs have been affected by several architectural vulnerabilities over the years. Vulnerable components in the architecture may incur invalid states due to design or implementation errors from the manufacturer, causing unwanted behaviours like system hangs, shutdowns, or, in the worst case, undefined states possibly exploitable. For example, the F00F bug was triggered by an invalid opcode that led to the lock-up of the CPU until it was rebooted [6]. The FDIV bug is in this category as well, as it simply provides wrong results for specific operands provided to the floating-point division instruction [5]. Although well-known, these bugs are not the only architectural CPU vulnerabilities. Many architectural bugs were never documented but only mentioned in CPU erratas [29, 30]. The specification update for the 11^{th} generation of Intel CPUs (released 2020) already contains 73 errata. While many of these errata might not be exploitable, e.g., incorrect values reported or failure to resume correctly from sleep states, the missing details make it impossible to guarantee non-exploitability. Recent vulnerabilities that have been found mostly by researchers [12, 29, 30, 36, 45, 55, 57, 60] are exploitable, though. These vulnerabilities allow an attacker to crash a system [12, 30], leak data from parts of cache lines of a different security domain [29], modify computation results in a different security domain [36, 55, 57, 60], or change the control flow of a different application [45]. Although all these bugs are observable on the architectural level, understanding the root cause is often still difficult [42]. Moreover, while it is often not difficult to trigger the bugs, it is extremely difficult to exploit them in a reliable way that goes beyond a denial-of-service attack [12, 30, 42, 45].

Table 1: Classification of transient and architectural vulnerabilities according to CWE originally targeted at software vulnerabilities. CWE-441 has no architectural counterpart yet. ÆPIC Leak represents the architectural counterpart for CWE-665.

Vulnerability Type		Transient Vulnerability	Architectural Vulnerability
CWE-416	Use-after-free	ZombieLoad [63], RIDL [78], Fallout [2], Spectre-STL [19]	iTLB multihit [30]
CWE-441	Confused Deputy	SWAPGS [49]	-
CWE-119	Out-of-bounds Operation	Spectre-PHT [39], Spectre v1.1 [38], Meltdown-BND [4]	GPU cache-line leak [29]
CWE-843	Type Confusion	Foreshadow-VMM [83]	F00F bug [6]
CWE-682	Incorrect Calculation	LVI-FP [61]	Plundervolt [57], V0LTpwn [36], VoltJockey [60], FDIV bug [5]
CWE-362	Race Condition	Meltdown [48], Foreshadow [72]	AMD Ryzen IRETQ bug [12]
CWE-691	Insufficient CF Management	Spectre-BTB [39], Spectre-RSB [41, 50]	Skylake bug [45]
CWE-74	Improper Neutralization (Injection)	LVI [73]	SEVerity [55]
CWE-665	Improper Initialization	CrossTalk [62], Medusa [54]	ÆPIC Leak (**this paper**)

Transient Vulnerabilities. Transient vulnerabilities are not directly visible on the architectural level, as they affect the microarchitecture. Observing these vulnerabilities requires a side channel [4]. Well-known transient vulnerabilities include Spectre [39] and Meltdown [48]. Meltdown-type attacks exploit delayed exception handling in out-of-order execution, while Spectre-type attacks leverage branch mispredictions. To leak data from the transient domain, the secret data is encoded into microarchitectural elements not cleared upon discarding transient instructions and transferred to the architectural state via a covert channel [1, 18, 47, 64, 81, 84]. As these vulnerabilities require indirect observation, they are much harder to detect accidentally. Similar to architectural vulnerabilities, many of them might not be exploitable [4]. However, as with architectural vulnerabilities, several of these vulnerabilities have been exploited successfully [1, 2, 19, 38, 39, 41, 48, 50, 61–63, 72, 73, 78, 83]. These vulnerabilities allow an attacker to read architecturally inaccessible data from the own process [19, 39], change the transient control flow of processes [38, 39, 41, 50], inject data into the transient domain [61, 73], and leak data from different security domains [2, 48, 62, 63, 72, 78, 83]. The last generation of Intel CPUs (Sunny-Cove-based CPUs) is not vulnerable to Meltdown-type attacks due to in-silicon mitigations.

3.2 Classification of Vulnerabilities

For software (and now also hardware) vulnerabilities, there is the CWE (Common Weakness Enumeration) classification. This classification contains more than 900 categories of vulnerabilities [70]. Intuitively, one would assume that the hardware vulnerabilities cover CPU vulnerabilities. However, while they are indeed classified in the hardware-bug categories in the CWE, sometimes even with their own categories, we argue that these categories are not necessary to enumerate the vulnerabilities. Looking at modern CPUs, they are designed using hardware-description languages (HDLs) [71]. Hence, CPUs can, to some extent, also be considered as *software*.

Our analysis shows that the underlying root causes of CPU vulnerabilities are not so different from (complex) software vulnerabilities. Thus, they can be classified using the existing software-vulnerability categories (cf. Table 1). This classification works both for architectural and transient vulnerabilities.

Out-of-bounds Operation (CWE-119). The transient-execution attack Spectre-PHT [4, 39] can be classified under CWE-119 "Improper Restriction of Operations within the Bounds of a Memory Buffer". The description of this category states: "The software performs operations on a memory buffer, but it can read from or write to a memory location outside of the intended boundary of the buffer." [70], which is precisely what is happening in Spectre-PHT, except that it is not the software but the CPU. Meltdown-BND [4] exploits a similar problem, where the hardware transiently ignores the bounds check for a buffer. Although there are not many details, the architectural vulnerability Intel SA-00219 [29] also fits into this category. On affected CPUs, the integrated graphics card has an incorrect bounds check that allows reading the first 64 bit of a cache line used inside SGX enclaves.

Use after Free (CWE-416). According to Schwarz et al. [63], the root cause of the transient-execution attacks known as microarchitectural data sampling (MDS) [2, 63, 78] is a use-after-free vulnerability in internal CPU buffers. The old content of these internal buffers, *i.e.*, the line-fill buffer and the store buffer, is used transiently in a faulting load, although the entry was already free'd by a previously finished load (or store). Similarly, in Spectre-STL [19], the CPU uses old stale memory locations that should have already been overwritten by newer stores, *i.e.*, it reads from a resource that was already "released". The iTLB multihit vulnerability [30] is an architectural instance of a use-after-free vulnerability. In this vulnerability, the CPU tries to use an old TLB entry that is not valid anymore, while a newer valid TLB entry already exists for the virtual address. Hence, although the old entry should have been released by creating the new entry, the CPU still tries to use the released one, leading to a CPU lockup [30].

Confused Deputy (CWE-441). A confused deputy vulnerability sees an intermediary forwarding a request to a target resource without preserving information about access permissions of the origin source. When the SWAPGS instruction is speculatively executed in kernel mode, it swaps the kernel GS register with the user GS register during the transient window. The transient swap causes the CPU to use user-provided values in the GS register [49]. The SWAPGS instruction acts as a confused deputy to the instructions dereferencing GS, leaving no trace of the origin of the GS value that was coming from userspace and not kernelspace. We did not identify any corresponding architectural vulnerability in this category.

Type Confusion (CWE-843). In the Foreshadow-VMM [83] variant of Foreshadow [72], the CPU suffers from a type confusion in the page-table entry of a guest page table. On a non-present fault inside the VM, the CPU treats the page-table entry like a host page table, interpreting the stored page frame number as a host physical address instead of a guest physical address. The F00F bug [6] can also be considered as a type confusion: the CPU locked the bus as it confused the register access of the opcode with a memory access, preventing the bus lock from being released as the CPU did not observe the completed memory access.

Incorrect Calculation (CWE-682). The LVI-FP vulnerability [61] shows that the transient result of floating-point values can be modified in certain corner cases where the operation requires a microcode assist. While the calculation is corrected architecturally, subsequent code that is executed transiently works with incorrect values. The FDIV bug is the famous example of an architectural incorrect calculation, where the result of floating-point divisions was incorrect for specific operands [5].

Race Condition (CWE-362). The first Meltdown-type attacks Meltdown-US [48] and Foreshadow [72] can be considered race conditions. In both cases, the data is already accessed and forwarded to dependent operations before the CPU realizes that the virtual address points to architecturally inaccessible data. While there are not many details available about the AMD Ryzen IRETQ bug [12], it is very likely a race condition, as it can only be triggered when executing the `iretq` instruction on one hyperthread, while running a CPU-bound loop on the other hyperthread [12]. In this setup, the hyperthread executing the `iretq` stalls until the sibling hyperthread pauses.

Insufficient Control-Flow Management (CWE-691). For both Spectre-BTB [39] and Spectre-RSB [41, 50], an attacker can change the transient control flow unexpectedly. As the CPU does not properly distinguish between different applications for branch-prediction targets, an attacker can inject an arbitrary branch target. On Intel Skylake CPUs, there is an architectural vulnerability that is not well understood but has similar effects [45]. Using 8-bit registers in a tight loop on one hyperthread can lead to unexpected changes of the instruction pointer on the sibling hyperthread.

Improper Neutralization (*Injection*) (CWE-74). LVI [73] injects values into a victim's transient data stream. In these attacks, the CPU does not properly neutralize the input to a faulting (or assisting) load, forwarding unrelated attacker-controlled data, *i.e.*, dependent operations receive incorrect data. This matches the description of CWE-74: "The software constructs all or part of a command, data structure, or record using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify how it is parsed or interpreted when it is sent to a downstream component." [70]. On AMD, there is an architectural vulnerability in this category called SEVerity [55]. Due to missing integrity checks of encrypted memory, an attacker can inject code into SEV-protected VMs.

Improper Initialization (CWE-665). The CrossTalk [62] transient-execution attack exploits the improper initialization of the internal staging buffer of the CPU. This buffer is used for the hardware random-number generator, as well as for the `cpuid` instruction. In both cases, only a part of the buffer is used, and the remaining part of the buffer is not cleared. However, the entire buffer is transmitted to the line-fill buffer, from where the improperly-initialized buffer can be leaked via RIDL [78] or ZombieLoad [63]. In this paper, we show the first architectural vulnerability in this category. We show that the reserved part of the APIC registers on Ice Lake and Alder Lake CPUs are not properly initialized, leaking stale data that was loaded from or stored to the LLC cache.

3.3 Missing Architectural Counterpart Discovery

Except for CWE-665 (Improper Initialization) and CWE-441 (Confused Deputy), we identified both transient and architectural vulnerabilities in every category in Table 1. We target the blank spot in CWE-665 by systematically analyzing the possible targets for architectural vulnerabilities caused by improper initialization. We focus on *data loads* where the underlying data structure is larger than the loaded data. For this, we focus on the I/O address space. As data leakage from valid memory addresses would have already been discovered, we do not expect any architectural vulnerabilities there. Similarly, previous work investigated the address space of model-specific registers [13] without discovering any data leakage.

In our experimental setup, we iterate over the entire I/O address space by mapping the address space page-by-page into the user space. Similarly to the approach described by Moghimi et al. [54], we groom microarchitectural buffers on the hyperthread while reading from the I/O address space. The grooming application simply reads and writes known data, ensuring that they end up in the store buffer, fill buffers, and cache hierarchy. If a value read from the I/O address space matches the known data, the physical address is reported as a potential source of data leakage.

Such a scan takes around 3 h to 4 h depending on the system we tested. On all Ice Lake and Alder Lake CPUs, this scan reported a physical address that architecturally leaks data

Table 2: Subset of tested CPUs and whether they are vulnerable (✓) or not (✗) to ÆPIC Leak. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Core i3-1005G1	Ice Lake	Sunny Cove	✓
Core i5-1035G1	Ice Lake	Sunny Cove	✓
Core i7-10510U	Comet Lake	Skylake	✗
Core i5-1135G7	Tiger Lake	Willow Cove	✗
Core i9-12900K	Alder Lake	Sunny Cove	✓
Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓

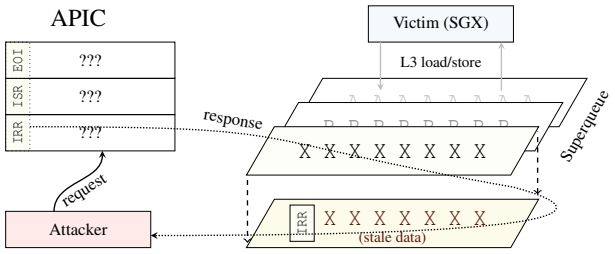

Figure 1: ÆPIC Leak reads a reserved part of an APIC register. The APIC uses the superqueue between L2 and LLC to transfer the data to the core. The reserved parts do not overwrite the superqueue entry, exposing stale values from previous reads and writes of other applications to the attacker.

from the sibling hyperthread: `0xFEE00000`. In Section 4, we provide an analysis of this architectural information leakage, showing that it is indeed caused by improper initialization. The scanning also led to several crashes, e.g., when reading from the Serial IO GPIO host controller. As scanning is only possible from ring 0, *i.e.*, the kernel, we do not consider this behavior security-relevant. While reading from address `0xFEE00000` is also only possible from the kernel, such an attacker is valid when attacking SGX enclaves.

4 ÆPIC Leak Overview

In this section, we introduce ÆPIC Leak, an architectural vulnerability in Intel CPUs that exploits undefined behavior in the APIC to leak data from the cache hierarchy. We provide an overview of ÆPIC Leak in Section 4.1, its threat model in Section 4.2 and analyze the root cause in Section 4.3. Based on the analysis, we introduce required building blocks for exploitation in Section 4.4.

4.1 Attack Overview

Figure 1 shows a high-level overview of ÆPIC Leak. ÆPIC Leak leaks values by *architecturally* reading the undefined range of APIC registers from ring 0, *i.e.*, the OS. Accessing bytes 4 to 15 of each 16 −byte register results in undefined behavior according to Intel [28]. This undefined behaviour includes reading either zeros or *0xFF*, system hangs, or triple faults on most CPUs. However, as discovered via the I/O address-space scan (cf. Section 3.3), this is not the case on

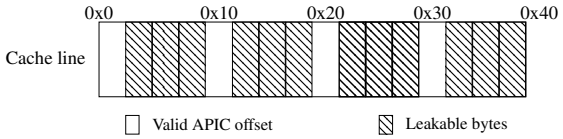

Figure 2: Leakable bytes in a 64-byte cache line.

Sunny-Cove-based CPUs. Instead, stale data from the superqueue is returned. Accessing any defined or undefined register in the byte-range 4-15, with a load width between 1 and 4 bytes, returns such stale data. Hence, ÆPIC Leak can atomically leak a 32-bit value per read. Load widths of 8 bytes or more return *0xFF*, and thus do not leak data.

The uninitialized data returned from ÆPIC Leak is not restricted to any security domain, *i.e.*, the origin can be user-space applications, the kernel, and, most importantly, SGX enclaves. Our hypothesis is that the invalid offsets in APIC registers are not properly initialized, *i.e.*, zeroed. Our experiments indicate that the superqueue is used as a temporary buffer for APIC requests. The superqueue entry contains stale data of recent memory loads and stores that traveled from the L2 to the L3 or the other direction. The APIC only overwrites the architecturally-defined parts of the register and leaves the stale values in the reserved part.

There is no correlation between the APIC register used for leaking data and the leaked data. Reading any reserved address within the APIC range `0xFEE00000-0xFEE003FF` leads to the same leakage. The only control over the leaked data is the cache-line offset. The cache-line offset of the used APIC address always matches the cache-line offset of the leaked data, which is also the case for MDS attacks [2, 63, 78].

As valid APIC register parts overwrite the stale value, the leakage pattern is as illustrated in Figure 2. For every 16 B block, the first 4 B contain valid APIC data, followed by 12 B stale data. Hence, ÆPIC Leak deterministically leaks 48 B from a cache line. Another limitation is that ÆPIC Leak only leaks even cache lines, *i.e.*, cache lines that start at an address that is a multiple of 128. As cache line pairs are typically transferred in pairs [27], we hypothesize that the second cache line is transferred first and then immediately overwritten by the first cache line, leaving only the stale data of the first cache line in the superqueue. Still, the leakage of ÆPIC Leak covers 37.5 % of any page. Section 5 shows that this is sufficient to, e.g., extract AES-NI keys from SGX enclaves, and presents different techniques to circumvent this limitation. However, this is exactly what we propose to leverage, to mitigate ÆPIC Leak at the software level (cf. Section 6.3).

4.2 Threat Model

Following most microarchitectural attacks on Intel SGX, we assume the attacker can execute privileged native code on the target machine. At the hardware level, we assume a Sunny-Cove-based Intel CPU (e.g., 10^{th} and 12^{th} generation code

name "Ice Lake" and "Alder Lake" and 3rd generation Xeon scalable "Ice Lake SP"). These CPUs are not vulnerable to any Meltdown-type attacks, such as Meltdown [48], Foreshadow [72,83], RIDL [78], or ZombieLoad [63]. ÆPIC Leak observes memory operations inside an Intel SGX enclave. We assume either a malicious hypervisor targeting secrets in guest enclaves or a privileged attacker willing to extract secrets from local enclaves, e.g., bypassing private contact discovery on Signal Servers [51], leaking DRM secrets or even SGX attestation keys. ÆPIC Leak only requires the OS or hypervisor to access the physical Local APIC to leak secrets, with no difference between the two settings. The attacker is either running on the same physical core, either on the sibling logical core or on the same logical core, e.g., if hyperthreading is disabled. While SGX enclaves can detect if hyperthreading is enabled during remote attestation [23], there is no recommendation to disable hyperthreading on CPUs with silicon fixes against Meltdown-type attacks. Thus, on Sunny-Cove-based CPUs, hyperthreading can be enabled. Still, even without hyperthreading, ÆPIC Leak can leak memory operations inside an SGX enclave, just with a reduced leakage rate.

In line with the SGX threat model, an attacker can rely on arbitrary operating-system features, such as the modification of page-table entries [76], the precise interrupts of enclaves using timer interrupts [74], or the execution of privileged SGX instructions, such as `EWB` to evict EPC pages.

Virtualized Environments. A malicious virtual machine with access to the host Local APIC could exploit ÆPIC Leak to observe data from other tenants or the hypervisor. However, no hypervisor we analysed exposes direct access to the host Local APIC. Usually, the APIC MMIO region, when enabled, is emulated by the hypervisor by intercepting the accesses to the region and managing the virtual interrupts [69]. In case Intel APIC virtualization (Intel APICv [28]) is enabled, the physical CPU emulates APIC functionality for the virtual CPUs in dedicated pages. We empirically verified that ÆPIC Leak does not work with APIC virtualization and APICv mode to leak from a guest VM. Thus, ÆPIC Leak does not allow guest virtualized systems to leak data. On the contrary, a malicious hypervisor could leverage ÆPIC Leak to leak secrets from guest VMs, leveraging its own Local APIC, irrespective of the guest APIC configuration.

Other Vendors and CPUs. We tested all Intel Core microarchitectures from Sandy Bridge (2nd generation) to Alder Lake (12th generation), and AMD CPUs from Zen to Zen 3. We did not discover any vulnerable CPU other than the ones based on Sunny Cove. Table 2 reports a subset of the CPU we tested, see Appendix C for the full list. We observe hangs or reads of *0x00* or *0xFF* on unaffected CPUs.

4.3 Leakage Analysis

In this section, we analyze the leakage of ÆPIC Leak, *i.e.*, from which microarchitectural element the data originates. We designed several experiments that show how the leakage source of ÆPIC Leak is different from previous microarchitectural attacks and demonstrate that ÆPIC Leak allows picking up stale values from the superqueue. We performed our tests on an Ice Lake Core i5-1035G1 machine, with Ubuntu 20.04.1, kernel 5.4.0-96, and the last microcode update installed (cf. Table 2).

4.3.1 Ruling out Microarchitectural Elements.

As we cannot directly observe from which microarchitectural element ÆPIC Leak leaks, we instead rule out microarchitectural elements from which ÆPIC Leak does not leak, expanding and systematizing the methodology from Schwarz et al. [63]. Our methodology is general to be applied to the study of the leakage of other CPU bugs. In this section, we describe the experiments we designed for all microarchitectural elements that are not involved, *i.e.*, where ÆPIC Leak still leaks the targeted data after clearing or circumventing them.

L1 Data Cache. By flushing the L1D via `MSR_0x10B` [23] and disabling hyperthreading, we ensure that the targeted data is not stored in the L1 while being leaked.

Line-Fill Buffer and Load Ports. We use the software sequences provided by Intel [34] to clear intermediate buffers, including the LFB and load ports, and still leak values.

L1 Instruction Cache. ÆPIC Leak leaks code and data, which travel through different paths in the hardware (e.g., code does not go through the LFB). It is unlikely that both paths (L1D and L1I) are affected and we see some combined leakage. Thus, we rule out the L1 cache and its line fill buffer.

Store Buffer. ÆPIC Leak is not limited to store operations but also leaks memory loads. Thus, we can eliminate the store buffer as leakage source. Moreover, ÆPIC Leak cannot leak *transient* stores which are only stored in the store buffer [2].

L2 Cache. ÆPIC Leak cannot leak data that is kept in L2 and not evicted towards L3. Thus we can exclude stale data in L2 as the source of leakage.

L3 Cache. ÆPIC Leak does not leak data kept in L3 while being exclusively used by other cores, and thus, not loaded towards the local L2 cache. This rules out all CPU caches.

Ring Bus. ÆPIC Leak cannot leak values processed by the GPU or from LLC slices exclusively used by other physical cores, also ruling out the ring bus. Moreover, ÆPIC Leak also works on Xeon CPUs without a ring bus [15].

Staging Buffer. ÆPIC Leak does not leak values from `cpuid` or `rdrand`, ruling out the staging buffer.

Memory. We also rule out the *DRAM* and *memory controller* by marking a memory region as *uncachable* to ensure that every store and load circumvents the cache hierarchy. ÆPIC Leak does not leak these loads and stores.

System Agent. As ÆPIC Leak does not leak values from *PCI* devices, we exclude this subsystem as leakage source.

Our experiments rule out the known internal buffers up to the L2 cache and the components in the uncore subsystem. Thus, we hypothesize that the leakage source is the internal buffer between the L2 and LLC cache, *i.e.*, the **superqueue**.

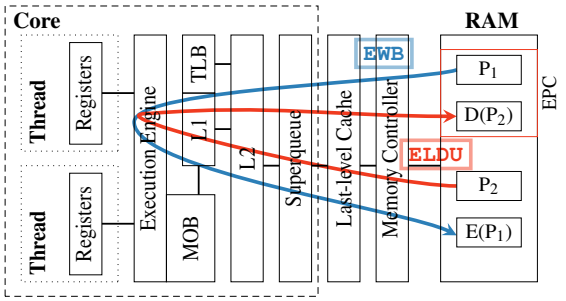

Figure 3: When executing the `ewb` instruction, a transparently encrypted page gets re-encrypted and moved to the non-EPC main memory. During this process, the unencrypted content of the page *flows* through the cache hierarchy. Vice versa for decryption with the `eldu` instruction.

We achieve the best leakage when building eviction sets that evict data from L2 but not from L3, and when relying on cache-line bouncing [52]. In both cases, the data deterministically moves through the superqueue between L2 and L3.

4.3.2 Performance Counter Analysis.

ÆPIC Leak does not trigger an architectural fault when performing a load instruction, even on reserved and undefined offsets of the MMIO region. However, we observed subtle microarchitectural differences when performing a load from a 16 B-aligned offset, whether it contains defined data or has been reserved by the specification. For every load to a reserved or undefined region, we observe a higher latency (437 cycles (n = 1000, $\sigma_{\bar{x}}$=0.57)) in contrast to a valid offset (47 cycles (n = 1000, $\sigma_{\bar{x}}$=0.03)). The `MACHINE_CLEARS.COUNT` performance counter indicates that the invalid load triggers an exception not forwarded to the architectural level. Furthermore, `OFFCORE_REQUESTS_OUTSTANDING.X` performance counters indicate that the core sends offcore requests as the loads are not satisfied by the local APIC, increasing the `CYCLE_ACTIVITY.STALLS_X` and, hence, the observed access time. As the data is not in any cache (and the PTE marked uncachable), the miss creates an entry in the superqueue and allocates a line fill buffer [46]. Table 5 in Appendix C gives an overview of all performance counters that show differences for defined and undefined offsets.

4.4 Building Blocks

Reading from a reserved part of the APIC does not provide any control over which cache line is leaked. However, we introduce three building blocks to influence which cache line is leaked. First, we force the target lines into the superqueue. Second, we increase the leakage of the specific target line. Finally, we extract the target line from the noisy measurements.

Forcing Data into the Superqueue. To target specific data, ÆPIC Leak first forces target cache lines into the superqueue. Van Bulck et al. [72] demonstrated that `ewb` and `eldu` swapping instructions bring plaintext data into the L1 cache while moving EPC pages. As the EPC pages are copied from memory to the L1 cache, and back from the L1 cache to memory, the–at this point unencrypted–data also travels through the superqueue. Hence, an attacker can bring data from an arbitrary enclave page into the superqueue using these instructions (cf. Figure 3). ÆPIC Leak uses a modified version of the Linux SGX driver to identify the enclave coupled with a target process and continuously swaps the target pages of the enclave. There is no need for ÆPIC Leak to run in the same or parallel hyperthread of the victim process, as the EPC swapping mechanism works independently from the enclave owning the page. Furthermore, as the EPC memory is persistent during the enclave's existence, ÆPIC Leak does not require the enclave to be active, *i.e.*, executing ECALLs during the leak. We refer to this technique as Enclave Shaking. Direct integration into the driver allows targeting an arbitrary enclave on the system, including Intel's quoting enclave.

Increasing the Leakage. While Enclave Shaking forces data from an EPC page into the superqueue, there is no control over which cache line is leaked. To solve that problem, we introduce Cache Line Freezing, a novel technique that provides control over which cache line of the page is leaked. Cache Line Freezing exploits that the cache access pattern of the hyperthread running in parallel to the attacker influences which entry is used, and thus leaked, from the superqueue.

Counterintuitively, Cache Line Freezing continuously accesses a page offset x of unrelated pages to *increase* the probability of leaking the cache line at the same offset x in the enclave. Specifically, the parallel hyperthread evicts the cache set of the target cache line with a crafted eviction set made up by continuously accessing several contiguous zero-filled pages at the same page offset of the target line. In our setup, it is sufficient to keep iterating over 256 virtually-contiguous pages at offset x, to trigger the effect.

The access pattern ensures that zero-filled cache lines with the same offset as the target cache line are continuously evicted from the L2 cache, and thus interact with the superqueue. Note that Cache Line Freezing does not work if the cache lines of the eviction set are set to values different from '0', or if their pages are all mapped to the kernel zero-page, *i.e.*, have never been written.

While the exact interaction of Cache Line Freezing with the superqueue is unknown, we hypothesize that zero-filled loads and stores are optimized. This would be in line with the observation that Ice Lake can eliminate stores of '0's to cache lines that only contain '0's [14]. Furthermore, we observe that if only one half of the cache lines of the eviction set is filled with zeros, we can only leak that same half of the target line with this method. This indicates that the granularity of this zero optimization is 32 B, in line with the memory bus width [27]. We assume that the zero data is marked differently in the superqueue, *i.e.*, only in the metadata without overwriting the entry. As a result, the eviction set keeps the entry in the superqueue used without overwriting it. Thus, stale data is

preserved over a longer duration, increasing the probability to leak it. Notice that Cache Line Freezing is the only building block of ÆPIC Leak where enabling hyperthreading has an impact. When hyperthreading is disabled, Cache Line Freezing must interleave with the attacker process on the same logical core, reducing its efficacy.

Cache Line Freezing allows an attacker to precisely select which cache line numbers to leak from the victim, and thus, to control which line is sampled. In case Cache Line Freezing is not leveraged, ÆPIC Leak would simply degrade to a sampling-based attack, similarly to MDS attacks, and additional techniques might be needed to reconstruct the original order of the leaked data [63, 78].

Extracting the Target Line. In addition to the content of the target cache line of the targeted EPC page (CL_{target}), unrelated values are leaked. Unrelated values include code and data involved in swapping pages and leaking values as well as data from general system activity. Since ÆPIC Leak is an architectural bug that deterministically reads stale data in the superqueue, the only noise it incurs is leaking such unrelated values. To filter these values, we establish a noise profile by leaking the content of the cache line with the same cache-line index from a different EPC page (CL_{noise}). Based on this noise profile, we can remove all values that have a similar frequency for CL_{noise} and CL_{target}. These values are likely independent from the content of the EPC page. Infrequent values that only occur for CL_{noise} or CL_{target} are likely secret-independent values from other applications or the OS and can thus also be ignored. The remaining values observed for CL_{target} are sorted by frequency. The value occurring with the highest frequency is likely the actual value of CL_{target}.

Due to the zero optimization (cf. previous paragraph), ÆPIC Leak cannot directly leak zero-filled blocks, as they are not stored in the superqueue. Instead, ÆPIC Leak can infer that a cache line contains zeros if there is not a single value with a distinct frequency, *i.e.*, the two most-frequent values have a similar frequency.

4.5 Performance Evaluation

To evaluate ÆPIC Leak's leaking characteristics, we set up a debug enclave that generates secret data via the `rdrand` instruction. This data is generated during an initial ECALL, and the page is targeted with ÆPIC Leak. To verify the correctness of the leaked data, we use the `edbrg` to read the generated page from the debug enclave *after* the leakage to ensure no other source is contributing to the leakage.

Repeating each cache line leak 2000 times, we achieve a leakage rate of 334.8 B/s with an average error rate of 7.8 % (n = 100, σ = 2.4 %). Decreasing the number of repetitions to 200 , the leakage rate increases to 1.76 kB/s with an average error rate of 16.0 % (n = 100, σ = 4.1 %) due to the increased noise of unrelated values. Note that in contrast to transient-execution attacks, all leaked values are correct. Noise only refers to data of other applications. Due to ÆPIC Leak limitations (cf. Section 4.1), this approach leaks 37.5 % of a page. This percentage can be further extended by combining different exploitations techniques, as we show in Section 5.

5 ÆPIC Leak Exploitation

In this section, we describe three attacks leveraging ÆPIC Leak against SGX enclaves. While in theory, ÆPIC Leak could leak memory from VMs or the hypervisor, no major hypervisor maps the host APIC MMIO region in the guest. We evaluate our attacks on the Ice Lake Core i5-1035G1.

5.1 Attack Techniques

We describe two attack techniques with ÆPIC Leak. We either target the data section of an enclave to leak secret *data at rest*, or we target the SSA area to leak *data in use* in the registers. Due to the limitations of which cache-line parts ÆPIC Leak can leak (cf. Section 4.1), the most effective technique to leak the target secret depends on the victim application. We observe no difference while leaking data from debug, pre-release or release enclaves.

Leaking Data and Code Pages. The straightforward use case for ÆPIC Leak is to combine Enclave Shaking and Cache Line Freezing to leak the data (and code) at rest of an SGX enclave. With Enclave Shaking and Cache Line Freezing, we target every cache line of a target page to leak 48 B of each even cache line within the page. This results in an overall leakage rate of 37.5 % of the page content. We repeat this process for each enclave page to recover a memory dump of an enclave. This technique is usable while the enclave is not running, resulting in a consistent state of the enclave data.

Leaking Register Values. Although ÆPIC Leak only leaks values from the superqueue, we can also use it to leak register values. During an asynchronous event, e.g., an interrupt, the hardware stores the current enclave registers in the SSA. Hence, the current register values are stored in the EPC. From there, we can again use Enclave Shaking and Cache Line Freezing to target a specific cache line containing one of the enclave registers and partially reconstruct the value of this register. Furthermore, by combining ÆPIC Leak with SGX-step [74], we can precisely single step the enclave, interrupting the enclave after each instruction. Hence, leaking the partial register state is possible after each executed instruction. As ÆPIC Leak does not require the enclave to run, we can target the SSA page with no timing restrictions, potentially recovering a full register trace of the enclave. However, due to the leakage limitations, ÆPIC Leak is restricted to the registers specified in Table 3.

Based on the register leakage, we identify a generic technique to leak data copied inside enclaves: the `__memcpy` function uses the `rdi` register as temporary storage to move data from the source over `rdi` to the destination. Since ÆPIC Leak can leak the upper 32 bit of the `rdi` register, this allows leaking 50 % of any data copied with `__memcpy` inside enclaves.

Table 3: Leakable SSA registers. For underlined GP-registers (e.g., rdi) ÆPIC Leak can only leak the upper 32-bit as the lower 32-bit are overshadowed by valid APIC registers.

Class	Registers
General Purpose	rdi r8 r9 r10 r11 r12 r13 r14
SIMD	xmm0-1 xmm6-9

5.2 Breaking AES-NI

Our first attack targets the 128-bit key in the constant time AES encryption provided by the Intel IPP library [26]. The IPP library leverages AES-NI for cryptographic primitives. The AES-NI primitives are tightly entangled with the enclave execution to, e.g., unseal and seal data or transfer data outside the enclave. We use the provided AES example from the official IPP GitHub repository [33]. The example uses the `ippsAESInit` function to initialize the AES context and the `ippsAESDecryptCTR` function to decrypt data with the AES counter mode. Leaking the key is possible either if it is at rest in the data page, or if it is in use in a register.

Key on Data Page. We can dump all the enclave pages after the secret key is transferred to the enclave and resides in memory. If the attacker knows the memory offset where the key is stored, this offset can be targeted directly. Given that there is no ASLR in enclaves [66], and the code is typically not confidential [7], this is a realistic assumption. Depending on the enclave memory layout, this technique has an ad-hoc probability of 50 % to leak the key: if it is stored in an even cache line, extracting the key is possible, if it is stored at an odd cache line it cannot be leaked. In the latter case, an attacker can leak the key when it is in use.

Key in SIMD Register. We assume that the IPP primitives used in an enclave are usually not modified by an enclave developer. Therefore, we can find the functions leaking the key without analyzing the remaining enclave. Furthermore, we assume that the enclave code is not encrypted. For encrypted code, we could first either leak the decryption key, or simply the decrypted code.

We developed an sgx-gdb [22] script that traces a debug version of the target enclave. This script prints the content of all leakable registers listed in Table 3, which are stored in the SSA. We identified that the `k0_aes_DecKeyExpansion_NI` function, which is independent of the AES implementation, temporarily stores the AES key in the `xmm1` register. Hence, by interrupting the enclave during that function, ÆPIC Leak can leak 96 bit of the AES key from the SSA. We can recover the remaining 32 bit of the key in the `k0_aes128_KeyExpansion_NI` function. Furthermore, we also leak 96 bit of the initial value over the `xmm0` register in the `k0_EncryptStreamCTR32_AES_NI` function. The remaining 32 bit can also be easily bruteforced, as it is exactly known which bits are missing. On the i9-12900K, we can evaluate on average 403 million AES keys per second. Hence, in the worst case, it takes 10.7 s to bruteforce the missing bits.

Evaluation. We evaluate ÆPIC Leak with 100 different random keys and try to leak the AES keys with a single run of the attack. A full key recovery takes on average 1.35 s (n = 100, σ = 15.70 %) with a success rate of 94 %. In the remaining 6 cases, we leaked unrelated data from different applications. However, as an attacker can typically restart enclaves arbitrarily often, as it is the case with the Quoting Enclave, the attack can simply be repeated until the correct key is leaked.

5.3 Breaking RSA

To show that ÆPIC Leak is not limited to secrets in single registers, we target RSA keys from the IPP library reference example [33]. The enclave contains the secret primes P and Q as well as the private key parts dQ, dP and $qInv$. The public modulus N is computed in `ippsRSA_SetPrivateKeyType2` when initializing the RSA context.

Key on Data Page. Similar to AES-NI, we can target the memory used to store the RSA key parts. RSA keys are usually not stored directly in registers and are larger than 128 bit. Therefore, dumping the enclave data pages already has a high chance to leak parts of the stored RSA key.

Key in GP Registers. We can leak the RSA primes P and Q during the calculation of the public modulus N. The bits of the prime numbers P and Q temporarily flow through `r10` in the function `k0_cpDec_BNU`, and dP and dQ in the function `k0_ippsRSA_SetPrivateKeyType2`, and thus can be leaked.

Evaluation. We target RSA-1024 and leak 100 random 512 bit RSA primes with ÆPIC Leak. Leaking one of the secret parameters is already sufficient to fully recover all the remaining parameters and decrypt data. We count the attack on RSA as successful if we can fully recover at least one of the four RSA parameters. Leaking the parameters from registers has a success rate of 72 %. The attack takes on average 81.81 s (n = 100, σ = 48.92 %). In 18 cases, the parameters are not leaked as single-stepping the target instruction fails. In 10 cases, we leak data from other processes. However, the attack can typically be repeated until the correct key is leaked.

5.4 Breaking SGX Attestation

As previous work [72, 77, 79], we demonstrate leakage of sealing keys. With the sealing keys, it is possible to unseal sealed data as well as to decrypt the attestation keys, the fundamental security primitives used in SGX. The derivation process to get access to such a key is done in hardware with the `egetkey` instruction [7]. We can target the results of this instruction within the SGX implementation `sgx_unseal_data`. This function uses the generated `egetkey` key to derive the AES round keys used to unseal the encrypted data.

To test the attack, we build and debug an enclave that uses the `sgx_seal_data` and `sgx_unseal_data` functions to seal and unseal enclave data. By tracing the occurences of

the `enclu` instruction with `rax=1` we can precisly target the `egetkey` instruction. By following the hardcoded addresses to this instruction, we can find the `sgx_get_key` function without additional debug information. We use the offset of the `sgx_get_key` function to monitor its accesses and start ÆPIC Leak after observing the first access within our target enclave. From this point, we partially leak the `xmm0` and `xmm1` registers with Enclave Shaking and Cache Line Freezing and attack the AES key expansion as demonstrated in Section 5.2. We decrypt the sealed data passed to the untrusted environment with the extracted sealing key.

Extracting the EPID Private Key. We attack the official Intel quoting enclave [83] by modifying the untrusted sgx-psw aesmd service. The service handles the inter-process communication between the various Intel enclaves. In the modified service, we target the first call to the `verify_blob` function, which passes the encrypted EPID private key blob, retrieved from the provisioning enclave to the quoting enclave. During this ECALL, we use ÆPIC Leak to extract the blob's sealing key as described above. We use the extracted key together with the known zeroed initial value to decrypt the blob with `sgx_rijndael128GCM_decrypt`, and successfully verified the tag: as the GCM decryption is authenticated, this proves that the key is correct. Extracting the EPID keys allows an attacker to forge remote attestations, breaking the whole SGX system, as enclaves can then be emulated. Thus, SGX could not be trusted anymore on any platform until the keys are replaced. In addition, such an attack may also break TDX confidentiality, which bases its attestation on SGX [31].

6 Mitigations

In this section, we discuss mitigations in hardware (Section 6.1), firmware (Section 6.2), and software (Section 6.3).

6.1 Hardware

As a long-term solution, ÆPIC Leak has to be fixed in hardware. Given that older Intel CPU microarchitectures are not affected by ÆPIC Leak, we assume that fixing the issue in silicon is not complex. Similar to uninitialized variables in software, it might be sufficient to set the most-significant 12 B of any APIC to a defined value, such as '0' or '-1'. When accessing the APIC using 64 bit reads, the return for all reads is already '-1', regardless of whether the address points to a valid or reserved part of any APIC register. Hence, such functionality to return properly-initialized data already exists.

6.2 Firmware

In addition to hardware, mitigations can also be deployed on the firmware level, *i.e.*, as microcode update. Based on mitigations for other CPU vulnerabilities [34], we suspect that mitigations on the firmware level are the most promising mid-term solutions until the hardware is fixed. Intel can deploy such firmware fixes as microcode updates distributed and applied by the OS. As the microcode security version number is part of the SGX attestation [7], enclaves can refuse to run if the microcode updates are not applied. We propose three approaches to mitigate ÆPIC Leak name in microcode with different advantages and disadvantages.

Disable SGX. Even if disabling SGX is not a real mitigation, ÆPIC Leak only targets SGX enclaves, thus, microcode can simply disable SGX. Without SGX, there is no target within the threat model of ÆPIC Leak. Coincidentally, Intel deprecated SGX on Ice Lake and Alder Lake client CPUs [32]. However, ÆPIC Leak is still relevant, as Intel did not deprecate SGX on server CPUs (e.g., Ice Lake SP). Thus, while exploiting an enclave with ÆPIC Leak would not be possible on client CPUs (preventing widespread exploitation), it is still possible by using server CPUs. Leaking the Intel keys on a single up-to-date machine is sufficient to break the SGX ecosystem, as these keys can be used to emulate attestation. Hence, ÆPIC Leak must also be mitigated on server CPUs.

Enforce x2APIC. ÆPIC Leak exploits that the legacy xAPIC and not the x2APIC is used on most systems. One of the main differences between xAPIC and x2APIC is the interface. The xAPIC is accessed using memory-mapped I/O (MMIO). This is the interface exploited with ÆPIC Leak. In contrast, the x2APIC does not support the MMIO interface for performance reasons [21]. Instead, the communication interface of the x2APIC is based on MSRs. We verified that the MMIO range is indeed disabled when enabling x2APIC, fully preventing ÆPIC Leak. Reads from the MMIO range when x2APIC is enabled return -1.

The x2APIC specification [21] states that switching from x2APIC to xAPIC is only possible by disabling the local APIC unit. As this can only be done by writing to the `IA32_APIC_BASE` MSR, a microcode update could enable the x2APIC at boot and prevent the disabling of the x2APIC. An enforced x2APIC is supported by Linux (tested on Ubuntu 20.04.1, kernel 5.4.0-96), and ensures that ÆPIC Leak cannot be mounted. This solution is the only firmware-based solution that does not incur any performance penalties. In case enforcing x2APIC mode would not be possible in microcode, an alternative solution would be to insert the APIC mode in the attestation process. The enclave attestation may simply fail if x2APIC mode is not enabled. As a positive side effect to fully mitigating ÆPIC Leak, enforcing x2APIC might even slightly improve the system performance.

Disable Caching for EPC. The EPC range is by default marked as write-back memory using a memory-type range register (MTRR). A microcode update could easily change the memory type for the EPC range to uncachable. As shown in Section 4.3, ÆPIC Leak cannot leak load or stores to uncachable memory. Hence, an uncachable EPC range would fully prevent ÆPIC Leak. Costanet al. [8] also proposed an uncachable EPC range to protect enclaves against cache attacks. While SGX explicitly supports an uncachable EPC range, it is not clear whether the memory type is part of the

attestation [8]. Moreover, setting the entire EPC range to uncachable leads to a huge performance impact for enclaves, as no part of an enclave can benefit from caching anymore.

Flush Caches on EEXIT. As ÆPIC Leak leaks values traveling through the cache hierarchy, a possible mitigation is to flush all caches on an enclave exit (EEXIT). However, this is only sufficient if hyperthreading is disabled. With enabled hyperthreading, ÆPIC Leak can leak values from the enclave while it is running. Flushing caches with the same limitation, *i.e.*, disabling hyperthreading, is also the state-of-the-art mitigation for L1TF [23] and MDS attacks [25] on affected CPUs. The state of hyperthreading is already included in the attestation. Hence, SGX enclaves can also refuse to run if hyperthreading is enabled on the system.

We verified that the already-existing `wbinvd` successfully prevents the leakage if hyperthreading is disabled. The `wbinvd` instruction invalidates all cache levels and writes modified values back to the main memory. While we did not see any leakage after invalidating all cache levels, the invalidation is not very efficient. On average, we measured 321 655 cycles ($n = 5000$, $\sigma_{\bar{x}} = 406.3$) for executing the instruction. As this invalidation is required on every EEXIT, this mitigation has a huge impact on ECALL and OCALL latency. However, we expect that as the L1 flush MSR [23], Intel can implement an L2 flush MSR. As ÆPIC Leak is limited to leaking data moving between L2 and LLC, an LLC flush might not be necessary. While a huge performance overhead, such a buffer flushing was also used for L1TF and MDS attacks.

6.3 Software

As the OS or hypervisor are untrusted, mitigations implemented there are ineffective. However, mitigations can be implemented into the trusted software part of the SGX ecosystem, such as the enclave itself, or indirectly via the attestation.

Secret Alignment. A limitation of ÆPIC Leak is that the first 4 bytes of every 16-byte block cannot be leaked. Hence, we propose a software solution that splits secrets and stores the parts of the secrets only in these non-leakable 4 bytes. Our software workaround is similar to the Intel-proposed workaround for SA-00219 [29]. For CPUs affected by SA-00219, no secrets can be stored in the first 8 bytes of a cache line. Hence, Intel added functionality to the SGX SDK to misalign buffers, ensuring that they do not start at the beginning of a cache line. For ÆPIC Leak it is more complicated, as only 4 consecutive bytes can be used, in contrast to the 56 bytes for SA-00219. We propose to rely on AVX scatter and gather instructions to automatically spread a secret over memory such that only the non-leakable parts of memory are used. Listing 1 (Appendix A) shows a sample proof-of-concept implementation for 128-bit secrets, such as AES-NI keys. By relying on the scatter and gather instruction for single-precision floats, these functions can spread 4-byte blocks over one cache line. As the source and destination memory addresses are 64-byte aligned, all parts of the secret are overshadowed by valid APIC registers, hence there is no leakage.

This software workaround protects data at rest, as well as the loading from and storing to memory. However, there is still a small remaining attack surface left. When the secrets are already loaded to CPU registers, they are spilled to main memory on an (asynchronous) enclave exit. As this is done by the hardware, there is no possibility for the software to protect the secrets at this point. Thus, if an attacker triggers such an exit in the short time window where the secrets are in the CPU registers, the attacker can leak up to 96 bits of the secrets via the SSA. However, as ÆPIC Leak can only leak up to 96 bits of every *even* cache line, there are parts of the SSA that cannot be leaked (cf. Section 4.1). Hence, if only `xmm{2-5}` and `xmm{10-15}` are used for (round) keys, AES-NI can still be used securely inside an enclave.

Transient Secrets. As ÆPIC Leak leaks secrets that are moved between the L2 and the LLC cache, a possible software mitigation could also ensure that secrets never leave the CPU registers and the L1 cache. Previous work showed that it is possible to implement cryptographic algorithms, e.g., AES, by only using CPU registers [56]. However, in an enclave setting, an attacker can arbitrarily interrupt an enclave with high precision [74], forcing every register to be stored to main memory. Enclaves cannot opt-out from storing certain registers in the SSA [28]. Hence, the only workaround is to ensure that secrets are never architectural. With Mimosa, Guan et al. [17] leveraged hardware transactional memory to ensure secrets never leave the private cache and cannot be spilled to memory. Unfortunately, hardware transactional memory is not available on Sunny-Cove-based CPUs. As a more obscure variant, an enclave could leverage speculative execution to only work on secrets in the transient domain [80]. However, this would require an enclave to mount side-channel attacks to make the computed results visible. Listing 2 (Appendix B) shows a sample code for realising this software workaround for AES encryption.

7 Conclusion

We presented ÆPIC Leak, the first architectural CPU vulnerability that allows leaking values from the cache hierarchy. ÆPIC Leak works on the newest Intel CPUs based on Ice Lake, Alder Lake, and Ice Lake SP and does not rely on hyperthreading enabled. ÆPIC Leak enables attacks against SGX enclaves on Ice Lake CPUs, forcing specific data into caches and leaking targeted secrets. We show attacks that allow leaking data held in memory and registers. We demonstrate how ÆPIC Leak completely breaks the guarantees provided by SGX, deterministically leaking AES secret keys, RSA private keys, and extracting the SGX sealing key for remote attestation. We finally propose several firmware and software mitigations that would prevent ÆPIC Leak from leaking sensitive data or completely prevent ÆPIC Leak.

Acknowledgments

We thank the anonymous reviewers and our shepherd Alessandro Sorniotti, for their guidance, comments and their valuable feedback. Funding for part of this research was provided by generous gifts from Amazon and Red Hat. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandt ner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[2] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[3] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*, 2020.

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[5] Tim Coe. Inside the pentium-fdiv bug. *Doctor Dobb's Journal*, 1995.

[6] Robert R Collins. The intel pentium f00f bug description and workarounds. *Doctor Dobb's Journal*, 1997.

[7] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[8] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al. *Secure processors part II: Intel SGX security analysis and MIT sanctum architecture*. Now Publishers, 2017.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[10] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. Smashex: Smashing sgx enclaves using exceptions. In *CCS*, 2021.

[11] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security Symposium*, 2019.

[12] Matt Dillon. Some Ryzen Linux Users Are Facing Issues With Heavy Compilation Loads, 2017. URL: https://www.phoronix.com/forums/forum/hardware/processors-memory/955368-some-ryzen-linux-users-are-facing-issues-with-heavy-compilation-loads/page7#post955498.

[13] Christopher Domas. Hardware Backdoors in x86 CPUs. *Black Hat US*, 2018.

[14] Travis Downs. Ice Lake Store Elimination, 2020. URL: https://travisdowns.github.io/blog/2020/05/18/icelake-zero-opt.html.

[15] Bahaa Fahim, Yen-Cheng Liu, Chung-Chi Wang, Donald C. Soltis, Terry C. Huang, Tejpal Singh, Bongjin Jung, and Nazar Haider. Shared mesh, 2017. US Patent 2017/0019350 A1.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*, August 2020.

[17] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *S&P*, 2015.

[18] Lorenz Hetterich and Michael Schwarz. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*, 2022.

[19] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[20] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *S&P*, 2016.

[21] Intel. Intel 64 Architecture x2APIC Specification, 2008.

[22] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[23] Intel. L1 Terminal Fault SA-00161, 2018. URL: https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[24] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[25] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[26] Intel. Developer Reference for Intel Integrated Performance Primitives Cryptography, 2019. URL: https://software.intel.com/en-us/ipp-crypto-reference.

[27] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[28] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2019.

[29] Intel. INTEL-SA-00219, November 2019. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00219.html.

[30] Intel. Machine Check Error Avoidance on Page Size Change, 2019. URL: https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/advisory-guidance/machine-check-error-avoidance-page-size-change.html.

[31] Intel. Intel Trust Domain Extensions, 2021. URL: https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf.

[32] Intel. 12th Generation Intel Core Processor Family, 2022. URL: https://cdrdv2.intel.com/v1/dl/getContent/655258.

[33] Intel. Intel Integrated Performance Primitives Cryptography, 2022. URL: https://github.com/intel/ipp-crypto.

[34] Intel. Microarchitectural Data Sampling, 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html.

[35] Trent Jaeger. Operating system security. *Synthesis Lectures on Information Security, Privacy and Trust*, 2008.

[36] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[37] Hareesh Khattri, Narasimha Kumar V Mangipudi, and Salvador Mandujano. Hsdl: A security development lifecycle for hardware technologies. In *HOST*, 2012.

[38] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[40] Altug Koker, Thomas A. Piazza, and Murali Sundaresan. Scatter/gather capable system coherent cache, 2016. US Patent 9,471,492 B2.

[41] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[42] Mathias Krause. Watch Your Step(ping): Atoms Breaking Apart, 2021. URL: https://grsecurity.net/watch_your_stepping_atoms_breaking_apart.

[43] Tsvika Kurts, Zelig Wayner, and Tommy Bojan. Apparatus and method for bus signal termination compensation during detected quiet cycle, 2007. US Patent 7,227,377 B2.

[44] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. Hacking in Darkness: Return-oriented Programming against Secure Enclaves. In *USENIX Security Symposium*, 2017.

[45] Xavier Leroy. How I found a bug in Intel Skylake processors, 2017. URL: http://gallium.inria.fr/blog/intel-skylake-bug/.

[46] David Levinthal. Performance analysis guide for intel core i7 processor and intel® xeon 5500 processors, 2009.

[47] Moritz Lipp, Daniel Gruss, and Michael Schwarz. Amd prefetch attacks through power and time. In *USENIX Security Symposium*, 2022.

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

[49] Andrei Lutas and Dan Lutas. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. In *BlackHat Europe*, 2019.

[50] G. Maisuradze and C. Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*, 2018.

[51] Moxie Marlinspike. Technology preview: Private contact discovery for signal, 2017.

[52] Paul E McKenney, Jonathan Appavoo, Andi Kleen, Orran Krieger, Rusty Russell, Dipankar Sarma, and Maneesh Soni. Read-copy update. In *AUUG Conference Proceedings*, 2001.

[53] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[54] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[55] Mathias Morbitzer, Sergej Proskurin, Martin Radev, and Marko Dorfhuber. Severity: Code injection attacks against encrypted virtual machines. In *WOOT*, 2021.

[56] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR Runs Encryption Securely Outside RAM. In *USENIX Security Symposium*, 2011.

[57] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[58] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[59] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security Symposium*, 2021.

[60] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *Asian-HOST*, 2019.

[61] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *USENIX Security Symposium*, 2021.

[62] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*, 2021.

[63] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[64] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[65] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[66] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[67] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[68] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[69] Terenceli. Local APIC virtualization, 2018. URL: https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2018/08/29/apicv.

[70] The MITRE Corporation. Cwe version 4.6, 2021. URL: http://cve.mitre.org/.

[71] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[72] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[73] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[74] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[75] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[76] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[77] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Sgaxe: How sgx fails in practice, 2020.

[78] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[79] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions, 2020.

[80] Jack Wampler, Ian Martiny, and Eric Wustrow. Exspectre: Hiding malware in speculative execution. In *NDSS*, 2019.

[81] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated Discovery Of Microarchitectural Side Channels. In *USENIX Security Symposium*, 2021.

[82] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.

[83] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution, 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf.

[84] Yuval Yarom and Katrina Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[85] Michał Zalewski. American Fuzzy Lop, 2021. URL: https://github.com/Google/AFL.

A Software Workaround

Listing 1 shows a possible software workaround for specific scenarios, e.g., for protecting AES-NI keys. Secrets can be scattered to memory locations shadowed by valid APIC registers and thus not leakable. This is possible with a single AVX instruction. Furthermore, scattered secrets can also be copied into one XMM register with a single instruction. If an XMM register is chosen, that cannot be leaked via the SSA, this drastically reduces the attack surface. To ensure that a secret key cannot be leaked, this method can additionally be combined with the transient computation shown in Appendix B.

```
#define load_secret128(mem) _mm_i32gather_ps((float*)(
    mem), _mm_set_epi32(12, 8, 4, 0), 4)

#define store_secret128(mem, secret) _mm_i32scatter_ps
    ((float*)(mem), _mm_set_epi32(12, 8, 4, 0), secret,
     4)

#define define_secret128(name, secret) unsigned int
    __attribute__((aligned(64))) name[] = {((unsigned
    int*)(secret))[0], 0, 0, 0, ((unsigned int*)(
    secret))[1], 0, 0, 0, ((unsigned int*)(secret))
    [2], 0, 0, 0, ((unsigned int*)(secret))[3], 0, 0,
     0, }

// define an 128-bit secret at compile time
define_secret128(secret, "ABCDEFGHIJKLMNO");
// load an 128-bit secret from memory into an
    XMM register
__m128 xsecret = load_secret128(secret);
// store an 128-bit secret to memory
store_secret128(secret2, xsecret);
```

Listing 1: A software workaround for spreading 128-bit secrets to non-leakable 4-byte blocks using AVX.

B Transient AES Computation

Listing 2 demonstrates how AES-NI can be used securely inside SGX by doing all computations in the transient domain. During transient execution, e.g., via a mispredicted branch, the round key is loaded using the scatter-based technique shown in Appendix A. The round key is then used in the transient execution to encrypt the plain text, and encode one byte of the result in the cache. After the transient execution, this encoded byte can be inferred using a side-channel attack. The encoding can be chosen in a way that it can be decoded from within the enclave using, e.g., Evict+Reload, but not from outside the enclave, e.g., with Prime+Probe. While this method is not very efficient, it can protect an AES key even if the CPU is affected by ÆPIC Leak.

C Performance Counters and CPUs

Table 4 shows all tested CPUs for ÆPIC Leak. Table 5 shows performance counters that differ when loading from defined and undefined offsets of the local APIC MMIO region.

```
char aes_encrypt_get_byte(__m128 message, int
    target_byte) {
 char bytevalue[256 * 4096];
 // the following code is never executed
    architecturally, only transiently
 if(<mispredicted>) {
  // repeat for all AES-NI rounds
  __m128 roundkey = load_secret128(roundkey[i]);
  // AES-NI encryption rounds
  _mm_aesenc_si128(message, roundkey)
  [...]
  // encode in cache
  volatile char dummy = bytevalue[((message >> (8 *
    target_byte)) & 0xFF) * 4096];
 }
 // recover encoded byte from cache
 for(int i = 0; i < 256; i++)
  if(flush_reload(bytevalue[i * 4096])) return i;
}
```

Listing 2: The function transiently encrypts a message using AES-NI, encodes one chosen byte of the ciphertext in the cache, and recovers it from the cache using Flush+Reload. Although this function has to be called 16 times to get all bytes of the ciphertext, it ensures that registers containing key material never end up in the SSA.

Table 4: Tested CPUs that are (✓) or are not (✗) vulnerable. All tested Sunny-Cove-based CPUs are vulnerable.

CPU	Microarchitecture	Based on	ÆPIC Leak
Intel Core i5-2520M	Sandy Bridge	Sandy Bridge	✗
Intel Core i5-3230M	Ivy Bridge	Sandy Bridge	✗
Intel Core i3-4160T	Haswell	Haswell	✗
Intel Core i7-4790	Haswell	Haswell	✗
Intel Core i3-5010U	Broadwell	Haswell	✗
Intel Core i7-6700K	Skylake	Skylake	✗
Intel Core i3-7100T	Kaby Lake	Skylake	✗
Intel Core i3-8130U	Kaby Lake R	Skylake	✗
Intel Core i7-8565U	Whiskey Lake	Skylake	✗
Intel Core i7-8700K	Coffee Lake	Skylake	✗
Intel Core i9-9980HK	Coffee Lake	Skylake	✗
Intel Core i9-9900K	Coffee Lake	Skylake	✗
Intel Core i7-10510U	Comet Lake	Skylake	✗
Intel Core i3-1005G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1035G1	Ice Lake	Sunny Cove	✓
Intel Core i5-1135G7	Tiger Lake	Willow Cove	✗
Intel Core i9-12900K	Alder Lake	Sunny Cove	✓
Intel Xeon E5-1630 v4	Broadwell	Haswell	✗
Intel Xeon E3-1505M v5	Skylake	Skylake	✗
Intel Xeon E-2176M	Coffee Lake	Skylake	✗
Intel Xeon Silver 4208	Cascade Lake-SP	Cascade Lake	✗
Intel Xeon Platinum 8375C	Ice Lake SP	Sunny Cove	✓
Intel Celeron N3350	Apollo Lake	Goldmont	✗
Intel Celeron J4005	Gemini Lake	Goldmont Plus	✗
Intel Celeron N4500	Jasper Lake	Tremont	✗
AMD Ryzen 5 2500U	Zen	Zen	✗
AMD Ryzen 5 3550H	Zen	Zen	✗
AMD Ryzen Threadripper 1920X	Zen	Zen	✗
AMD Ryzen 7 3700X	Zen 2	Zen 2	✗
AMD Ryzen 7 5800X	Zen 3	Zen 3	✗
AMD EPYC 7443	Zen 3	Zen 3	✗

Table 5: Performance counter differences on loads to defined and undefined offsets of the APIC memory-mapped region on Intel i3-1005G1 CPU.

Performance Counter	Defined	Undefined
BR_INST_RETIRED.FAR_BRANCH	0.05 ($\sigma_{\bar{x}}$ = 0.00)	1.05 ($\sigma_{\bar{x}}$ = 0.00)
CORE_POWER.LVL0_TURBO_LICENSE	68.14 ($\sigma_{\bar{x}}$ = 0.11)	627.76 ($\sigma_{\bar{x}}$ = 0.09)
CPU_CLK_UNHALTED.DISTRIBUTED	68.13 ($\sigma_{\bar{x}}$ = 0.11)	627.73 ($\sigma_{\bar{x}}$ = 0.10)
CPU_CLK_UNHALTED.ONE_THREAD_ACTIVE	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CPU_CLK_UNHALTED.REF_DISTRIBUTED	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CPU_CLK_UNHALTED.REF_XCLK	0.75 ($\sigma_{\bar{x}}$ = 0.00)	7.08 ($\sigma_{\bar{x}}$ = 0.00)
CYCLE_ACTIVITY.CYCLES_L1D_MISS	191.84 ($\sigma_{\bar{x}}$ = 0.36)	176.34 ($\sigma_{\bar{x}}$ = 0.02)
CYCLE_ACTIVITY.CYCLES_MEM_ANY	866.04 ($\sigma_{\bar{x}}$ = 0.73)	2656.24 ($\sigma_{\bar{x}}$ = 0.12)
CYCLE_ACTIVITY.STALLS_L1D_MISS	427.20 ($\sigma_{\bar{x}}$ = 0.58)	2062.15 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_L2_MISS	235.36 ($\sigma_{\bar{x}}$ = 0.26)	1885.75 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_L3_MISS	235.36 ($\sigma_{\bar{x}}$ = 0.26)	1885.76 ($\sigma_{\bar{x}}$ = 0.20)
CYCLE_ACTIVITY.STALLS_MEM_ANY	1101.16 ($\sigma_{\bar{x}}$ = 0.93)	4541.89 ($\sigma_{\bar{x}}$ = 0.29)
CYCLE_ACTIVITY.STALLS_TOTAL	235.37 ($\sigma_{\bar{x}}$ = 0.26)	1886.16 ($\sigma_{\bar{x}}$ = 0.25)
DSB2MITE_SWITCHES.COUNT	0.95 ($\sigma_{\bar{x}}$ = 0.01)	5.91 ($\sigma_{\bar{x}}$ = 0.21)
DSB2MITE_SWITCHES.PENALTY_CYCLES	0.95 ($\sigma_{\bar{x}}$ = 0.01)	5.91 ($\sigma_{\bar{x}}$ = 0.21)
EXE_ACTIVITY.1_PORTS_UTIL	1.95 ($\sigma_{\bar{x}}$ = 0.02)	83.57 ($\sigma_{\bar{x}}$ = 0.02)
EXE_ACTIVITY.2_PORTS_UTIL	2.64 ($\sigma_{\bar{x}}$ = 0.02)	35.75 ($\sigma_{\bar{x}}$ = 0.02)
EXE_ACTIVITY.3_PORTS_UTIL	2.05 ($\sigma_{\bar{x}}$ = 0.01)	18.89 ($\sigma_{\bar{x}}$ = 0.01)
EXE_ACTIVITY.4_PORTS_UTIL	0.76 ($\sigma_{\bar{x}}$ = 0.00)	2.13 ($\sigma_{\bar{x}}$ = 0.00)
ICACHE_64B.IFTAG_HIT	3.58 ($\sigma_{\bar{x}}$ = 0.01)	20.80 ($\sigma_{\bar{x}}$ = 0.29)
ICACHE_64B.IFTAG_STALL	2.01 ($\sigma_{\bar{x}}$ = 0.00)	33.80 ($\sigma_{\bar{x}}$ = 1.21)
IDQ.DSB_CYCLES_ANY	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.DSB_CYCLES_OK	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.DSB_UOPS	6.89 ($\sigma_{\bar{x}}$ = 0.00)	24.52 ($\sigma_{\bar{x}}$ = 0.16)
IDQ.MITE_CYCLES_ANY	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.73 ($\sigma_{\bar{x}}$ = 0.14)
IDQ.MITE_CYCLES_OK	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.79 ($\sigma_{\bar{x}}$ = 0.15)
IDQ.MITE_UOPS	6.11 ($\sigma_{\bar{x}}$ = 0.00)	21.79 ($\sigma_{\bar{x}}$ = 0.15)
IDQ.MS_CYCLES_ANY	16.32 ($\sigma_{\bar{x}}$ = 0.12)	385.65 ($\sigma_{\bar{x}}$ = 0.19)
IDQ.MS_SWITCHES	16.32 ($\sigma_{\bar{x}}$ = 0.12)	385.65 ($\sigma_{\bar{x}}$ = 0.19)
IDQ.MS_UOPS	16.38 ($\sigma_{\bar{x}}$ = 0.12)	385.56 ($\sigma_{\bar{x}}$ = 0.19)
IDQ_UOPS_NOT_DELIVERED.CORE	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
IDQ_UOPS_NOT_DELIVERED.CYCLES_0_UOPS_DELIV.CORE	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
IDQ_UOPS_NOT_DELIVERED.CYCLES_FE_WAS_OK	19.84 ($\sigma_{\bar{x}}$ = 0.20)	411.07 ($\sigma_{\bar{x}}$ = 0.18)
INST_RETIRED.STALL_CYCLES	12.10 ($\sigma_{\bar{x}}$ = 0.00)	13.10 ($\sigma_{\bar{x}}$ = 0.00)
INT_MISC.ALL_RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}}$ = 0.01)	27.49 ($\sigma_{\bar{x}}$ = 0.01)
INT_MISC.CLEAR_RESTEER_CYCLES	0.25 ($\sigma_{\bar{x}}$ = 0.01)	25.32 ($\sigma_{\bar{x}}$ = 0.02)
INT_MISC.RECOVERY_CYCLES	0.31 ($\sigma_{\bar{x}}$ = 0.01)	27.19 ($\sigma_{\bar{x}}$ = 0.01)
INT_MISC.UOP_DROPPING	0.01 ($\sigma_{\bar{x}}$ = 0.00)	5.20 ($\sigma_{\bar{x}}$ = 0.00)
ITLB_MISSES.WALK_ACTIVE	0.94 ($\sigma_{\bar{x}}$ = 0.00)	10.76 ($\sigma_{\bar{x}}$ = 0.34)
ITLB_MISSES.WALK_PENDING	0.94 ($\sigma_{\bar{x}}$ = 0.00)	10.80 ($\sigma_{\bar{x}}$ = 0.34)
L1D_PEND_MISS.PENDING	23.98 ($\sigma_{\bar{x}}$ = 0.04)	22.06 ($\sigma_{\bar{x}}$ = 0.02)
L1D_PEND_MISS.PENDING_CYCLES	23.98 ($\sigma_{\bar{x}}$ = 0.04)	22.06 ($\sigma_{\bar{x}}$ = 0.02)
MACHINE_CLEARS.COUNT	0.01 ($\sigma_{\bar{x}}$ = 0.01)	1.10 ($\sigma_{\bar{x}}$ = 0.01)
MEM_INST_RETIRED.ALL_LOADS	2.95 ($\sigma_{\bar{x}}$ = 0.00)	3.95 ($\sigma_{\bar{x}}$ = 0.00)
MEM_INST_RETIRED.ANY	3.55 ($\sigma_{\bar{x}}$ = 0.00)	4.55 ($\sigma_{\bar{x}}$ = 0.00)
MEM_LOAD_RETIRED.L1_HIT	1.95 ($\sigma_{\bar{x}}$ = 0.00)	2.95 ($\sigma_{\bar{x}}$ = 0.00)
RESOURCE_STALLS.SCOREBOARD	57.34 ($\sigma_{\bar{x}}$ = 0.05)	458.50 ($\sigma_{\bar{x}}$ = 0.06)
RS_EVENTS.EMPTY_CYCLES	54.92 ($\sigma_{\bar{x}}$ = 0.05)	318.39 ($\sigma_{\bar{x}}$ = 0.06)
RS_EVENTS.EMPTY_END	54.92 ($\sigma_{\bar{x}}$ = 0.05)	318.39 ($\sigma_{\bar{x}}$ = 0.06)
TIME	44.05 ($\sigma_{\bar{x}}$ = 0.04)	433.64 ($\sigma_{\bar{x}}$ = 1.69)
TOPDOWN.BACKEND_BOUND_SLOTS	289.81 ($\sigma_{\bar{x}}$ = 0.25)	2346.41 ($\sigma_{\bar{x}}$ = 0.13)
TOPDOWN.BR_MISPREDICT_SLOTS	0.86 ($\sigma_{\bar{x}}$ = 0.06)	86.21 ($\sigma_{\bar{x}}$ = 0.05)
TOPDOWN.SLOTS_P	341.06 ($\sigma_{\bar{x}}$ = 0.54)	3138.51 ($\sigma_{\bar{x}}$ = 0.68)
UOPS_DECODED.DEC0	0.46 ($\sigma_{\bar{x}}$ = 0.00)	3.25 ($\sigma_{\bar{x}}$ = 0.10)
UOPS_DISPATCHED.PORT_0	2.90 ($\sigma_{\bar{x}}$ = 0.02)	47.32 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_1	2.86 ($\sigma_{\bar{x}}$ = 0.02)	57.18 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_2_3	3.05 ($\sigma_{\bar{x}}$ = 0.00)	9.05 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_DISPATCHED.PORT_4_9	4.30 ($\sigma_{\bar{x}}$ = 0.00)	6.30 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_DISPATCHED.PORT_5	2.86 ($\sigma_{\bar{x}}$ = 0.02)	42.42 ($\sigma_{\bar{x}}$ = 0.02)
UOPS_DISPATCHED.PORT_6	5.67 ($\sigma_{\bar{x}}$ = 0.03)	72.28 ($\sigma_{\bar{x}}$ = 0.03)
UOPS_DISPATCHED.PORT_7_8	4.25 ($\sigma_{\bar{x}}$ = 0.00)	7.25 ($\sigma_{\bar{x}}$ = 0.00)
UOPS_EXECUTED.CORE	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.72 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_1	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_2	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_3	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CORE_CYCLES_GE_4	25.83 ($\sigma_{\bar{x}}$ = 0.09)	242.76 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_1	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_2	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_3	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.CYCLES_GE_4	25.74 ($\sigma_{\bar{x}}$ = 0.09)	242.58 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.STALL_CYCLES	25.76 ($\sigma_{\bar{x}}$ = 0.09)	242.68 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_EXECUTED.THREAD	25.76 ($\sigma_{\bar{x}}$ = 0.09)	242.68 ($\sigma_{\bar{x}}$ = 0.09)
UOPS_ISSUED.ANY	29.38 ($\sigma_{\bar{x}}$ = 0.12)	272.26 ($\sigma_{\bar{x}}$ = 0.12)
UOPS_ISSUED.STALL_CYCLES	29.45 ($\sigma_{\bar{x}}$ = 0.12)	272.27 ($\sigma_{\bar{x}}$ = 0.12)
UOPS_RETIRED.SLOTS	29.60 ($\sigma_{\bar{x}}$ = 0.09)	249.53 ($\sigma_{\bar{x}}$ = 0.08)
UOPS_RETIRED.STALL_CYCLES	29.60 ($\sigma_{\bar{x}}$ = 0.09)	249.53 ($\sigma_{\bar{x}}$ = 0.08)
UOPS_RETIRED.TOTAL_CYCLES	28.00 ($\sigma_{\bar{x}}$ = 0.08)	247.95 ($\sigma_{\bar{x}}$ = 0.08)