Osiris: Automated Discovery of Microarchitectural Side Channels

Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow, *CISPA Helmholtz Center for Information Security*

https://www.usenix.org/conference/usenixsecurity21/presentation/weber

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Osiris: Automated Discovery of Microarchitectural Side Channels

Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, Christian Rossow
CISPA Helmholtz Center for Information Security

Abstract

In the last years, a series of side channels have been discovered on CPUs. These side channels have been used in powerful attacks, e.g., on cryptographic implementations, or as building blocks in transient-execution attacks such as Spectre or Meltdown. However, in many cases, discovering side channels is still a tedious manual process.

In this paper, we present Osiris, a fuzzing-based framework to automatically discover microarchitectural side channels. Based on a machine-readable specification of a CPU's ISA, Osiris generates instruction-sequence triples and automatically tests whether they form a timing-based side channel. Furthermore, Osiris evaluates their usability as a side channel in transient-execution attacks, *i.e.*, as the microarchitectural encoding for attacks like Spectre. In total, we discover four novel timing-based side channels on Intel and AMD CPUs. Based on these side channels, we demonstrate exploitation in three case studies. We show that our microarchitectural KASLR break using non-temporal loads, FlushConflict, even works on the new Intel Ice Lake and Comet Lake microarchitectures. We present a cross-core cross-VM covert channel that is not relying on the memory subsystem and transmits up to 1 kbit/s. We demonstrate this channel on the AWS cloud, showing that it is stealthy and noise resistant. Finally, we demonstrate Stream+Reload, a covert channel for transient-execution attacks that, on average, allows leaking 7.83 bytes within a transient window, improving state-of-the-art attacks that only leak up to 3 bytes.

1 Introduction

Since first described by Kocher [51] in 1996, side channels have kept challenging the security guarantees of modern systems. Side channels targeted mostly cryptographic implementations in the beginning [5, 37, 51, 69]. By now, they have also been shown to be powerful attacks to spy on user behavior [36, 67, 81]. Moreover, in transient-execution attacks, such as Meltdown [57] or Spectre [50], side channels are vital.

Side channels often arise from abstraction and optimization [79]. For example, due to the internal complexity of modern CPUs, the actual implementation, *i.e.*, the microarchitecture, is abstracted into the documented architecture. This abstraction also enables CPU vendors to introduce transparent optimizations in the microarchitecture without requiring changes in the architecture. However, these optimizations regularly introduce new side channels that attackers can exploit [3, 10, 56, 69, 74, 80, 86, 89].

Although new side channels are commonly found, discovering a side channel typically requires manual effort and a deep understanding of the underlying microarchitecture. Moreover, with multiple thousand variants of instructions available on the x86 architecture alone [1], the number of possible side effects that can occur when combining instructions is too large to test manually. Hence, manually identified side channels represent only a subset of the side channels of a CPU.

Indeed, automatically finding CPU-based side channels is challenging. Side channels consist of a carefully-chosen interplay of multiple orthogonal instructions that are syntactically far apart from each other. Typically, they require instructions that change an inner CPU state and others reading (leaking) this inner state. In addition, many side channels rely on specific instructions to reset the internal state to a known one. For example, the popular Flush+Reload side channel [101] flushes cache lines to reset the state, fills a secret-dependent cache line, and uses another cache access to leak the new state. Identifying such an interplay *automatically* is notoriously hard, fueled by thousands of CPU instructions, their possible combinations, and the lack of mechanisms to verify the existence of potential side-channel candidates.

Automation attempts, therefore, have focused on particular types of side channels so far. With Covert Shotgun and AB-Synthe, Fogh [27] and Gras et al. [30], respectively, automated the discovery of contention-based side channels. Their tools identified several side effects of instructions when run simultaneously on the two logical cores, *i.e.*, hyperthreads, of a physical CPU core. However, their approach does not generalize beyond contention-based side channels. Moghimi et al. [65]

considered the sub-field of microarchitectural data-sampling (MDS) attacks. Their tool, Transynther, combines and mutates building blocks of existing MDS attacks to find new attack *variants*. However, they do not try to find new classes of side channels, and only focus on cache-based covert channels.

In this paper, we present a generic approach to automatically detect timing-based side channels that do not rely on contention. We introduce a notation for side channels that allows representing side channels as triples of instruction sequences: one that resets the inner CPU state (*reset sequence*), one that triggers a state change (*trigger sequence*), and one that leaks the inner state (*measurement sequence*). Based on this notation, we introduce Osiris, an automated tool to identify such instruction-sequence triples. Osiris relies on fuzzing-like techniques to combine instructions of the targeted instruction-set architecture (ISA) and analyzes whether the generated triple forms a side channel. Osiris supports an efficient search scheme which can cope with side effects between different fuzzing iterations, a challenging phenomenon that is not present in most other fuzzing domains.

In contrast to CPU instruction fuzzing [20], Osiris does not search for undocumented instructions but instead relies on a machine-readable ISA specification. Such a specification exists for x86 [1] and ARMv8 [8]. As these specifications contain all ISA extensions as well, Osiris first reduces the candidate set to instructions that can be executed as an unprivileged user on the target CPU. From this candidate set, Osiris combines instructions and tests whether they can be used as a covert channel. In such a case, the found triple is reported as a covert channel, and thus also as a potential side channel. The current proof-of-concept implementation of Osiris is limited to finding timing-based single-instruction side channels in an unguided manner. However, even such a simple setup involves many challenges that require a careful design to enable finding interesting sequence triples.

We ran Osiris for over 500 hours on 5 different Intel and AMD CPUs with microarchitectures from 2013 to 2019. Osiris found both existing and novel side channels. The existing side channels include Flush+Reload [101], and the `AVX2` side channel described by Schwarz et al. [84]. Moreover, Osiris discovered four new side channels using the `RDRAND` and `MOVNT` instructions, as well as in the x87 floating-point and AVX vector extensions.

In three case studies, we demonstrate that these newly identified side channels enable powerful attacks. Based on the findings of non-temporal moves (`MOVNT`), we show FlushConflict, a microarchitectural kernel-level ASLR (KASLR) break that is not mitigated by any of the hardware fixes deployed in recent microarchitectures. We successfully evaluate FlushConflict on the new Intel Ice Lake and Comet Lake microarchitectures, where the performance is on par with previous microarchitectural KASLR breaks from which almost all stopped working on the newest microarchitectures. Furthermore, with the detected side-channel leakage of `RDRAND`, we show that we can build a fast and reliable cross-core covert channel that is also applicable to the cloud. Our cross-core covert channel can transmit 95.2 bit/s across virtual machines on the AWS cloud. We use these side channels as a covert channel in a Spectre and in a Meltdown attack to leak on average 7.83 B in one transient window.

In addition to the practical evaluation of the side channels, we demonstrate that our new primitives can evade detection via performance counters [19, 40, 48, 72], and even undermine the security of state-of-the-art proposals for secure caches [59, 76, 97]. Thus, this paper shows that side channels are quite versatile, making it hard to build robust detection methods that cover all possible side channels. We stress that it is important to build automated tooling for analyzing the attack surface to design more effective countermeasures in the future. Osiris is a first step, and even when limiting ourselves to single-instruction sequences, we show that many unknown side channels can be uncovered automatically.

To summarize, we make the following contributions:

1. We introduce an approach to automatically find timing-based microarchitectural side channels that follow a generic instruction-sequence-triple notation and develop a prototype implementation[1] for it.
2. We discover 4 new side channels on Intel and AMD CPUs.
3. We present FlushConflict, a microarchitectural KASLR break that works on the newest Intel microarchitectures, and a noise-resistant cross-core cross-VM covert channel that does not rely on the memory subsystem.
4. We analyze existing side-channel detection and prevention methods and show that they are flawed with respect to our newly discovered side channels.

Responsible Disclosure. We disclosed our findings to Intel on January 19, 2021, and they acknowledged our findings on January 22, 2021. Moreover, we disclosed the cross-core covert channel to AMD on February 5, 2021.

2 Background

In this section, we provide background for this work.

2.1 Microarchitecture

The microarchitecture refers to the actual implementation of an ISA. Typically, the microarchitecture is not fully documented, as it is transparent to the programmer. Hence, performance optimizations are often implemented transparently in the microarchitecture. As a result of the optimizations and the abstraction, there is often unintended leakage of metadata, which can be exploited in so-called microarchitectural attacks. The most prominent microarchitectural attacks are cache-based side channels [31, 37, 101] and transient-execution attacks [50, 57, 79].

[1] Osiris's source is available at https://github.com/cispa/osiris

2.2 Side- and Covert Channels

Information is transmitted through so-called *channels*. These channels are often intended to exchange information between two entities, e.g., network or inter-thread communication. Nevertheless, some channels are unintended by the designers, e.g., power consumption or response time. Attackers can use unintended channels to transmit information between two attacker-controlled entities. We refer to such a channel as a *covert channel*. Moreover, attackers can abuse the channel to infer inaccessible data if a victim unknowingly is the sending end. In this case, the channel is called a *side channel*.

Both side and covert channels exist in modern microarchitectures [28]. CPU caches are probably the most popular microarchitectural components that can be abused for side or covert channels [35, 37, 69, 101]. As CPU caches are shared among different threads and even across CPU cores, adversaries can abuse them in a wide range of attack scenarios [36, 53, 57, 60, 64, 68].

2.3 Transient Execution Attacks

As modern CPUs follow a pipeline approach, instructions might be executed out of order and are only committed to the architectural level in the correct order. To avoid stalling the pipeline, the processor continues precomputing even when a branch value or a jump target is unavailable, e.g., due to a cache miss. This is enabled through several prediction mechanisms that allow speculatively executing instructions. When the branch target is evaluated, speculatively executed instructions are allowed to retire only in the case of correct prediction. Otherwise, the speculatively executed instructions are squashed. Instructions that are not retired but leave microarchitectural traces are called transient instructions [17, 46, 57].

Spectre [50] is one class of transient-execution attacks exploiting speculative execution. By mistraining a branch predictor, an attacker can influence the transient control flow of a victim application. In the transient control flow, an attacker typically tries to encode application secrets into the microarchitectural state. Using a side channel, this encoded information is later transferred to the architectural state. Meltdown [57] is another class of transient-execution attacks, exploiting the lazy handling of exceptions. On affected CPUs, inaccessible data is forwarded transiently before the exception is handled. Transient execution attacks commonly use the cache to encode leaked secrets [17, 50, 52, 57, 61] but can also use other side channels [12, 56, 80, 84].

2.4 Fuzzing

Fuzzing is a software testing technique that aims at finding bugs in software applications [9, 18, 73, 78, 88]. A fuzzer typically generates a large number of test inputs and monitors software execution over these inputs to detect faulty behavior. Due to the huge input space, fuzzers typically search for inputs with a high probability of triggering a bug while avoiding uninteresting input. Fuzzers usually follow one of two different approaches for generating input [9, 13]. Mutation-based fuzzers usually start with an initial set of inputs (seeds), then generate further test input by applying mutations, e.g., splicing or bit flipping [9, 21, 41]. Grammar-based fuzzers exploit existing input specifications to generate a model of the expected input format. Based on this model, the fuzzer efficiently generates accepted input [13, 38, 70]. Moreover, fuzzing approaches can be clustered in two classes based on how they generate new or mutated input. While blind fuzzing randomly generates input based on a grammar of predefined mutations [21, 39], guided fuzzing uses the current execution to guide the generation of new input. These techniques aim to maximize a given metric [9, 18, 73, 103].

Most research efforts on fuzzing target software applications. Nonetheless, hardware fuzzing is becoming increasingly popular [20, 30, 65]. Sandsifter [20] presents a search algorithm that allows efficiently finding undocumented x86 instructions. It applies byte-code mutation to generate new instructions and checks whether the processor can decode the generated instructions. ABSynthe [30] allows automatically synthesizing a contention-based side channel for a target program. It uses fuzzing to find instruction sequences that generate distinguishable contention on secret-dependent code execution. Mutation parameters in ABSynthe include instruction building blocks, repetition number, and use of memory barrier. Hardware fuzzing has also been utilized to improve existing Meltdown attacks [100] or find new variants of these attacks [65], automate the search for Spectre gadgets [90], and identify cross-core transient-execution attacks [77].

3 High-level Overview of Osiris

In this section, we introduce a notation that captures timing-based side channels based on *instruction-sequence triples* (Section 3.1) before we describe the design of Osiris. Side channels not exploitable via timing differences are out of scope for Osiris. We discuss challenges when using this new notation to find side channels (Section 3.2). Finally, we showcase the big picture of our fuzzing framework (Section 3.3).

3.1 Side-Channel Notation

For detecting side channels, we first focus on detecting covert channels, as every side channel can also be used as a covert channel. Regardless whether timing-based covert channels are used as side channels or as covert channels in transient-execution attacks, they follow these three steps:

(1) In the first step, the attacker brings a microarchitectural component, abused by the attack, into a known state. For example, the attacker might flush or evict a cache line (e.g., Flush+Reload, Prime+Probe, Evict+Reload) or power down the `AVX2` unit. We call this known state the *reset state* (S_0).

Table 1: Existing timing-based side channels mapped to sequence triples and whether our approach can find it (●) or cannot find it (○). Reasons for failure are that multiple instructions are required (), side channel only works across hardware threads (), or specific operands are required (#).

Side channel	Seq_{reset}	$Seq_{trigger}$	$Seq_{measure}$	Osiris	Reason
AVX [84]	sleep	AVX2 instr.	AVX2 instr.	●	
Flush+Reload [101]	CLFLUSH	mem. access	mem. access	●	
Flush+Flush [35]	CLFLUSH	mem. access	CLFLUSH	●	
Flush+Prefetch [33]	CLFLUSH	mem. access	PREFETCH	●	
BranchScope [25]	cond. jump	cond. jump	cond. jump	●	
Evict+Reload [74]	mem. accesses	mem. access	mem. access	○	, (#)
Evict+Time [69]	mem. accesses	mem. access	mem. access	○	, (#)
Prime+Probe [74]	mem. accesses	mem. access	mem. accesses	○	, #
Reload+Refresh [14]	mem. accesses	mem. access	mem. accesses	○	, #
Collide+Probe [56]	mem. access	mem. access	mem. access	○	#
DRAMA [75]	mem. access	mem. access	mem. access	○	#
Port contention [7]	sleep	execute	execute (same HT)	○	

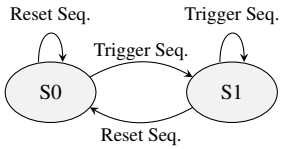


Figure 1: State machine representing different microarchitectural states and transitions between them.

We call a sequence of instructions that causes a transition to S_0 a *reset sequence* (Seq_{reset}).

(2) In the second step, the victim (or the sending end) changes the state of the abused microarchitectural component based on a secret. The victim might cache a value depending on the secret, or power up the AVX2 unit by executing an AVX2 instruction. We call the new state the *trigger state* (S_1). We call a sequence of instructions causing a transition to S_1 a *trigger sequence* ($Seq_{trigger}$).

(3) Finally, the attacker tries to extract the secret value by checking whether the abused component is in the reset state S_0 or the trigger state S_1. This is typically done by measuring the execution time of a particular instruction sequence, which we call the *measurement sequence* ($Seq_{measure}$). The measurement sequence may—in fact, typically *does*—have side effects beyond measuring, *i.e.*, it also influences the state.

Table 1 shows examples of these three instruction sequences for several known side channels. For example, Flush+Reload uses CLFLUSH as the reset, and memory accesses (e.g., via MOV) as trigger and measurement sequences. The careful reader will notice that existing side channels often do not require instruction *sequences*, but just a single instruction per step—a simplification that we will leverage ourselves later.

Figure 1 shows a state machine representing the relation between the three steps of an attack and the different microarchitectural states of the abused component. These two states could represent an abstraction over possibly more complex states of the component, e.g., different cache levels. However, to mount a side-channel attack, it is sufficient to distinguish and transit between two states only.

3.2 Challenges of Side-Channel Fuzzing

Based on this notation, we design Osiris, a fuzzer that aims to automatically find new microarchitectural side channels. The overall idea is to generate inputs, *i.e.*, instruction-sequence triples, and then detect whether such a triple forms a side channel. For this, Osiris executes a triple and measures the execution time of the measurement sequence. At an abstract level, we compare timings *with* and *without* prior execution of the trigger sequence. Large timing differences hint at side channels. While the overall idea is intuitive, several challenges complicate the search:

Unknown Sequences. First, as we aim for *novel* side channels, we cannot assume *a priori* knowledge of valid reset, trigger, or measurement sequences. This poses a significant challenge to fuzzing, as we have to fuzz all three inputs without knowing their relations. We are unaware whether an instruction sequence actually is a reset, trigger, or measure sequence. Even if we find a sequence (e.g., a trigger), we do not know which counterparts are required for the other two sequences (e.g., corresponding reset and measurement sequences).

Unknown Side Effects. Second, sequences on their own may have undesired side effects, such as measurement sequences that change the state. For example, memory accesses within the measurement sequence do not only passively observe the memory access time, but they also change the cache state. This implies that our state diagram becomes more complex, as measurement sequences may in fact act as triggers themselves. If we had a valid reset sequence, this would not be a problem, as we could revert this change. However, as mentioned above, we do not know the corresponding reset sequence, and therefore have to mitigate this problem conceptually.

Dirty State. Third, in the interest of efficiency, we want to fuzz as fast as possible. This, unfortunately, means that a subsequent sequence triple may inherit a dirty, non-pristine state from its successor. For example, if the first triple contains a memory access, the triple executed after that likely inherits the cache state. In other words, we cannot assume that sequence triples run in isolation. They do affect each other.

Generality. Fourth, we want to be as generic as possible and cover the entire instruction set of a given ISA. That is, instead of testing just a few popular instructions, we would like to explore the entire range of instructions and their combinations. To this end, we not only require knowledge of all instructions but also a semantic understanding of an instruction's syntax, such as its operands and their types.

Indistinguishability. Finally, executing similar instructions inevitably leads to similar, if not indistinguishable[2], side-channel candidates. In fact, we create thousands of sequence

[2] Indistinguishable side channels are those which lead to the same attacker observation on system states.

triples, many of which are close to each other. For example, with reference to known side channels, dozens of instructions use vector operations to power up the AVX unit. However, regardless of which instruction is executed, more or less the same side channel is found. Section 4 elaborates on how we solved these challenges for Osiris.

3.3 Big Picture

Figure 2 shows the big picture of Osiris, a fuzzer that tackles these challenges. In step (1), the *code generation stage*, we fuzz potential instruction sequences, *i.e.*, triples of $\mathsf{Seq}_{\mathsf{reset}}$, $\mathsf{Seq}_{\mathsf{trigger}}$, and $\mathsf{Seq}_{\mathsf{measure}}$. These sequences are generated from a machine-readable specification of the targeted architectures. The generated triples are then forwarded to step (2), the *code execution stage*. Here, the generated triples are executed in a special order (at least) twice—once including the trigger (*hot path*), and once without (*cold path*). We time the measurement sequence ($\mathsf{Seq}_{\mathsf{measure}}$) of both paths to see if the trigger sequence ($\mathsf{Seq}_{\mathsf{trigger}}$) causes timing differences. The timing difference is then processed in step (3). This *result confirmation stage* interprets a large timing difference as the first indicator of whether a given triple constitutes a side channel candidate. On top of this, to address many of the problems as mentioned earlier, there are additional validation routines that sort out actual side channels from wrong candidates. For example, we check whether (i) the reset sequence has any effect at all to exclude a bad triple combination, and (ii) a different fuzzing order confirms the result. Finally, in step (4), we feed the list of confirmed side channels to the *clustering stage*. This step clusters similar, indistinguishable side channels, to ease further analyses of the side channels.

4 Design and Implementation

Next, we discuss the implementation of Osiris for the x86 ISA and how we solved the challenges enumerated in Section 3.2. While we chose to implement and evaluate our fuzzer on this architecture, the overall design is equally applicable to processors that use a different instruction set, e.g., ARM processors. In the following, we present the implementation details for the four stages outlined in Figure 2.

4.1 Code Generation Stage

The goal of the code generation stage is to produce triples of assembly instruction sequences (a reset sequence $\mathsf{Seq}_{\mathsf{reset}}$, a trigger sequence $\mathsf{Seq}_{\mathsf{trigger}}$, and a measurement sequence $\mathsf{Seq}_{\mathsf{measure}}$). Since we are not aware of a clear feedback mechanism that can guide the creation of sequence triples, we opted for the creation of random x86 instructions. To bootstrap the code generation, we employ a grammar based on a machine-readable specification of x86 instructions. The code generation involves two phases: (1) an offline phase where all supported instruction sequences are generated, and (2) an online phase performing the creation of triples. The offline phase is executed once for each ISA and consists of instruction creation and machine-code file generation. The online phase is executed repeatedly for each run of the fuzzing process.

Table 2: Faulting instructions on Intel Core i7-9750H.

Signal	Number of Occurrences
Segmentation fault (*SIGSEGV*)	118
Floating-point exception (*SIGFPE*)	22
Illegal instruction (*SIGILL*)	10 508
Debug instruction (*SIGTRAP*)	1

4.1.1 Offline Phase

The output of the offline phase is an assembly file containing all possible instruction variants for the target ISA. This file is generated once and reduces the overhead required for generating and assembling instructions during runtime.

Generation of Raw Instructions. The first task is the generation of all valid x86 instructions. To achieve this, we leverage a machine-readable x86 instruction variant list from uops.info [1]. This list extends Intel's XED iForm[3] with additional attributes, e.g., effective operand size, resulting in a large number of instruction variants per instruction. For example, this list provides 35 variants for the mnemonic `MOV` and 26 variants for the mnemonic `XOR`, summing up to 14 039 x86 instruction variants overall. The list also contains comprehensive information about each instruction variant, e.g., extension or category, that we later use for the clustering.

Creation of the Machine Code. The second task is assembling the instructions to machine code. We try to reduce the number of instructions by treating all registers as equivalent, *i.e.*, Osiris does not generate the instruction with all possible register combinations. Osiris, w.l.o.g, relies on a fixed set of registers as operands for each instruction. We also exclude instructions that change the control flow (e.g., `RET`, `JMP`) as they may lead to an irrecoverable state. As branches have been studied extensively for microarchitectural attacks [3, 4, 6, 23–25, 50, 54], we do not assume that Osiris would find any new side channels for these instructions. Finally, we add a pseudo-instruction that allows idling the CPU for a certain period of time. This instruction is required to reset components that are based on power-saving features of the CPU, e.g., the `AVX2` SIMD unit. For each assembled instruction, the file also stores a set of attributes, e.g., the ISA extension or instruction category, that are used in the clustering phase.

[3] https://intelxed.github.io/ref-manual/xed-iform-enum_8h.html

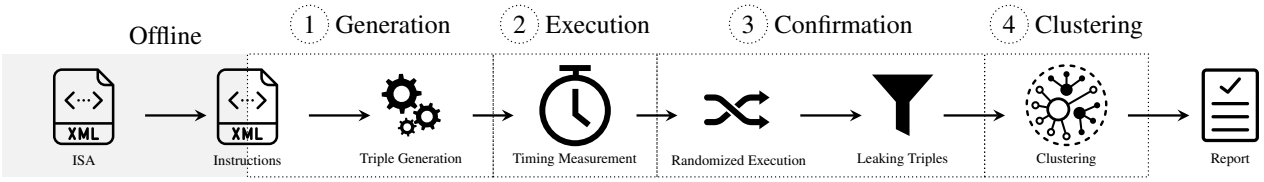


Figure 2: Overview of Osiris. The offline phase extracts available instructions from a machine-readable ISA description. The first phase generates sequence triples from these instructions. The execution phase measures their execution times and forwards triples with timing differences to the confirmation phase. If the timing difference persists on randomized execution of the triple, it is considered a side channel and forwarded to the clustering phase, which categorizes the triple and creates the final report.

4.1.2 Online Phase

When starting Osiris on a machine, the online phase first removes instructions that are not supported on the microarchitecture, and then generates all possible sequence triples.

Cleanup of Machine-Code File. The first task is the cleanup of the machine-code file generated in the offline phase. This is required since the generated machine-code file contains instruction variants for the entire x86 ISA, including all extensions. Hence, it contains a significant number of illegal instructions for a given microarchitecture. Moreover, the file may also include instructions that generate faults when executed by our framework, e.g., privileged instructions. The cleanup process is done by executing all instructions once and maintaining a list of all the instructions that terminated normally. This process reduces the number of instructions in the machine-code file considerably. For example, the number of user-executable instructions for an Intel Core i7-9750H is 3390, *i.e.*, 24.1 % of the instruction variants initially generated in the offline phase. Table 2 shows the distribution of faults generated in the cleanup process for this processor. The majority of the faults (98.7 %) are illegal-instruction faults, *i.e.*, the instruction is not supported at all or not in user space.

Generation of Sequence Triples. The second task is the generation of sequence triples from the list of executable instructions that are forwarded to the code execution stage. We exploit three observations that allow reducing the complexity of this task as well as the overhead of the fuzzing process:

1. Most existing non-eviction-based side channels require only one instruction in each of the sequences.
2. Idling the processor is used only as a reset sequence.
3. Trigger and measurement sequences may be formed of exactly the same instruction.

Consequently, in our implementation, the triples are generated by considering all possible combinations of single instructions, where the sleep pseudo-instruction is only used as a reset sequence. While our framework is easily extensible to support multi-instruction sequences, the search space quickly explodes—a topic we thus leave open to future work.

4.2 Code Execution Stage

The goal of the code execution stage is to execute generated input triples and analyze their outcome, *i.e.*, determine whether an executed triple forms a side channel.

Environment. The triple is executed within the process of Osiris to not suffer from the additional overhead of process creation. To reduce external influences, such as interrupts, Osiris relies on the operating system to reduce any noise. First, the operating system ensures that there are no core transitions that influence the measurement by pinning the execution of the triple to a dedicated CPU core. Additionally, this entire physical core is isolated to ensure that the code is unlikely to be interrupted, e.g., by the scheduler or hardware interrupts.

Setup. To measure the execution time of a triple, it is placed on a dedicated page in the address space between a special prolog and epilog. The prolog is responsible for saving all callee-saved registers according to the x86-64 System V ABI 2. The prolog furthermore ensures that the triple has one page of scratch space on the stack. Thus, there is no corruption if any of the instructions in the triple modifies the stack, e.g., the `POP` instruction. Furthermore, the prolog initializes all registers that are used as memory operands to the address of a zero-initialized writable data page. This prevents corrupting the memory of Osiris and ensures that executed instructions access the same memory page. Note that the zero-filled page is always the same, and the framework resets this page for every tested triple. The epilog is responsible for restoring the registers and the stack state, ensuring that any architectural change is reverted. Moreover, signal handlers are registered for all possible signals that can arise from executing an instruction, e.g., SIGSEGV. These handlers abort the execution of the current triple and restore a clean state for Osiris. Finally, we abstain from parallelization, as this could lead to unexpected interferences in shared CPU resources.

Measurement. Once the triple is prepared, Osiris executes the generated sequence twice, once with the trigger sequence $\mathsf{Seq}_{\mathsf{trigger}}$ (hot path) and once without (cold path), as illustrated in Figure 3. In both cases, the execution time of the measurement sequence $\mathsf{Seq}_{\mathsf{measure}}$ is measured. This code aims to detect the existence of a side channel by observing

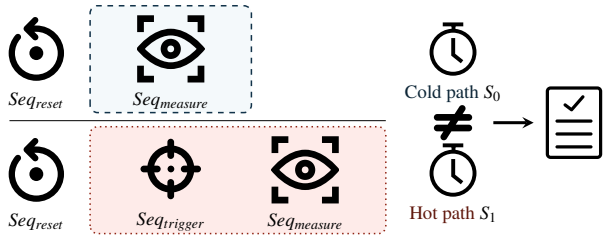


Figure 3: The execution stage receives the triple and executes $\mathsf{Seq}_{\mathsf{measure}}$ (cold path) and $\mathsf{Seq}_{\mathsf{trigger}}$, $\mathsf{Seq}_{\mathsf{measure}}$ (hot path) after $\mathsf{Seq}_{\mathsf{reset}}$. Timing differences for the two paths are reported.

timing differences in the measurement instruction, depending on whether or not a trigger was used. A significant difference between the two measurements indicates a candidate side channel that is then forwarded to the confirmation stage. To ensure precise time measurement and no unintentional dependency on the timing measurement itself, we add serializing and memory-ordering instructions around the measured code.

4.3 Result Confirmation Stage

The goal of the confirmation stage is to validate if a triple reported by the execution stage is an exploitable side channel. To confirm or refute these candidates, Osiris further analyzes the identified triples to rule out other side effects that could have led to the detected timing difference. Such side effects include unreliable reset sequences or a dirty state caused by previous execution (cf. Section 3.2). To eliminate non-promising candidates, we foresee the following mechanisms.

Repeated Execution. External factors, such as power-state changes or interrupts, can induce timing differences. To rule out such cases, Osiris executes the hot path and the cold path (cf. Section 3.3) over a predefined number of runs to compare the median of the timings for the two cases. In particular, this check is passed if the difference between the two medians is greater than a predefined threshold. The number of measurements is a parameter that allows setting a tradeoff between precision and runtime. While a high number of repetitions takes longer, it increases the confidence in the result, as external influences are statistically independent and thus average out. Too few repetitions reduce the confidence in the accuracy of the reported results, leading to false positives.

Non-Functional Reset Sequences. The initially observed timing difference may result from different sequence combinations leading to the desired state without actually performing the required transition. For example, consider a faulty reset sequence $\mathsf{Seq}_{\mathsf{reset}}$ that does not reset the state to S_0. A timing difference would still be detected by the first check if the test started in a state S_0. To ensure the correct functionality of $\mathsf{Seq}_{\mathsf{reset}}$, Osiris measures the execution time of $\mathsf{Seq}_{\mathsf{measure}}$ after the execution of $\mathsf{Seq}_{\mathsf{reset}}$. It then measures the timing after the execution of $\mathsf{Seq}_{\mathsf{trigger}}$ followed by $\mathsf{Seq}_{\mathsf{reset}}$. A negligible difference between the two measurements indicates that $\mathsf{Seq}_{\mathsf{reset}}$ actually resets the state to S_0 when triggered to S_1 by $\mathsf{Seq}_{\mathsf{trigger}}$. The check also implies that the state change observed in the first check must be caused by executing $\mathsf{Seq}_{\mathsf{trigger}}$. Consequently, the input formed of the sequence triple allows reaching the target, *i.e.*, it represents a potential side-channel.

Triple Reordering. Osiris executes all generated triples shortly after another. We may therefore experience undesired edge cases caused by dirty microarchitectural states and side effects caused by prior executions. We therefore test each sequence multiple times (twice in our evaluation), each time randomizing the order in which we test the fuzzed triples. We then ignore triples that do show discontinuous behavior in all tested permutations. This reordering ensures that we have a negligible probability that two given sequence triples are executed directly after each other in both runs, hence lowering the chances of repetitive dirty states being carried over.

Applicability in Transient Execution. Osiris also allows detecting whether a side channel can be used as covert channels for transient-execution attacks. To test the transient behavior of the side channel, Osiris executes $\mathsf{Seq}_{\mathsf{trigger}}$ speculatively using Retpoline as shown in previous work [87, 98]. We opted for this variant as it has a perfect misspeculation rate requiring no mistraining of any branch predictors [98]. Osiris allows to optionally enable this behavior in the confirmation stage.

4.4 Clustering Stage

Different sequence triples can lead to the detection of the same side channel. For example, for cache-based side channels, every instruction that accesses a memory address can act both as trigger and as measurement sequence. Due to the CISC nature of x86, many instructions explicitly (e.g., `ADD`) or implicitly (e.g., `PUSH`) access memory. Additionally, every instruction that flushes this address acts as a reset sequence. Similarly, in the `AVX2` side channel, different `AVX2` instructions can act both as trigger and as measurement sequence.

In the clustering stage, Osiris aims at clustering the input forwarded from the code execution stage into groups that represent different side channels. To achieve this, we can base our clustering on various properties of the involved instruction sequences. Examples of instruction properties include the instruction's extension, memory behavior, and the general instruction category (e.g., arithmetic or logical). Additionally, our tests showed that the timing difference tends to be an important clustering property. This procedure assumes that similar side channels show similarities in the properties of the corresponding instructions. We identify two categories of properties that can be used for clustering, as outlined next.

Static Properties. Triples can be classified based on properties of the contained instructions, such as the instruction category (e.g., *arithmetic* or *logical*) or the instruction extension (e.g., `AVX2` or `x87-FPU`). As this information is propagated from the instructions to the clustering phase, Osiris fundamen-

tally relies on this information for clustering. The clustering stage clusters the reported triples based on the instruction set extension of $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$. The intuition behind this clustering is that instruction-set extensions are strong indicators for the underlying microarchitectural root cause. Although this process cannot remove all duplicates, it significantly reduces the number of reported triples, thus, facilitating further analysis of the side channels.

Dynamic Properties. In addition to the static properties of instructions, it is also possible to cluster triples based on their dynamic effects. One of the dynamic properties Osiris supports for clustering is the observed timing difference. If multiple triples lead to the same timing difference, the root cause is likely the same, *i.e.*, access-time differences when accessing cached and uncached memory. Additionally, the clustering stage may cluster the triples based on their cache behavior. As shown by Moghimi et al. [65], performance counters can be used for clustering triples. By executing triples while recording performance counters, it is possible to dynamically observe which parts of the microarchitecture are active. This can also help to identify the root cause easier.

5 Results

In this section, we evaluate the design choices of Osiris based on the prototype implementation described in Section 4.

5.1 Evaluation Setup

We perform the fuzzing on 5 different CPUs and evaluate the case studies based on our results on a more extensive set of CPUs (cf. Table 4 and Table 5). We use a laptop with an Intel Core i7-9750H (Coffee Lake), and 4 desktop machines with an Intel Core i7-9700K (Coffee Lake), Intel Core i5-4690 (Haswell), AMD Ryzen 5 2500U (Zen), and AMD Ryzen 5 3550H (Zen+). All systems run Ubuntu or Arch Linux.

5.2 Performance

Before demonstrating Osiris's ability to find side channels, we evaluate its performance, *i.e.*, the number of triples tested per second. To measure this throughput, we first use the same instruction sequence for $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$. For the first measurement, we exclude the pseudo sleep instruction, as it—by construction—biases the code execution time. We only report the throughput for the oldest processor, *i.e.*, the Intel Core i5-4690. For this microarchitecture, there are 3377 instructions (after cleanup), leading to a total of $3377^2 = 11\,404\,129$ sequence triples. A full fuzzing run terminated in just 41 s, resulting in a throughput of 278 149 triples per second. To identify the bottleneck of our framework, we increased the number of repetitions of each triple from 1 to 10, *i.e.*, executed more code. In this experiment, the fuzzer took 127 s to complete (89 796 triples per second), resulting in a runtime increase by factor 3 only.

When including the pseudo sleep instruction, the overall runtime grows to 56 s and 271 s for 1 and 10 repetitions, respectively. That is, the throughput reduces to 202 370 triples per second (or 42 044 for 10 repetitions). This is a 37 % slowdown compared to the first run that excluded sleeping. Intuitively, sleeps imply that the fuzzer spends more time executing code. This explains the stronger impact of the actual code execution on the overall throughput compared to code generation. Increasing the number of repetitions by 10x, therefore, decreases the number of tested triples by a factor of 4.8. For the actual fuzzing run, $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$ are different. Hence, the number of sequence triples increases to $3377^3 = 38\,511\,743\,633$, leading to a runtime of nearly 5 days.

5.3 Clustering

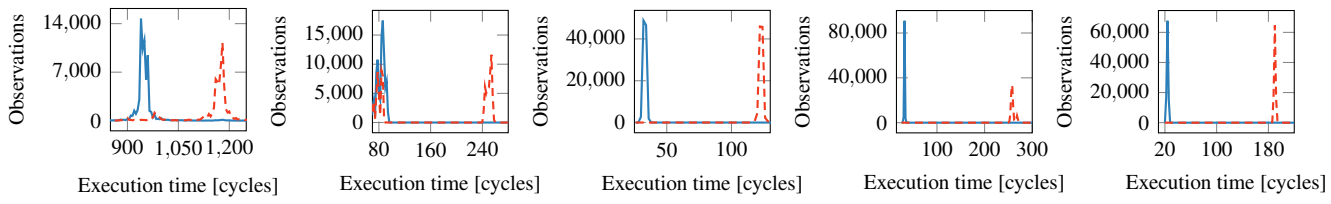
On the tested microarchitectures, Osiris successfully clustered the reported instances into fewer than 30 clusters. On the Intel i7-9750H, the 68 597 reported side channels were first clustered into 186 clusters. To further reduce the number of clusters caused by one side-channel variant, Osiris also provides the clustering based only on $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$, as these sequences contain the instructions causing the leakage. Based on these two sequences, the number of clusters is only 16. Table 7 (Appendix A) shows the numbers for other CPUs.

5.4 Rediscovering Known Side Channels

A typical test for software fuzzer is the rediscovery of old bugs, e.g., by searching for vulnerabilities in poorly tested software, checking for well-known CVEs, or uncovering bugs reported by prior work. Osiris also rediscovered two well-known side channels, Flush+Reload [101] and the `AVX2`-based side channel [84], as described in the following. Section 7 discusses some of the known side channels Osiris did not rediscover and provides the reason for that.

Flush+Reload-Based Side Channel. Osiris detects a total of 18 799 triples that can be classified as a variant of Flush+Reload. These triples have in common that $\mathsf{Seq}_{\mathsf{reset}}$ is in either `CLFLUSH` or `CLFLUSHOPT`, and $\mathsf{Seq}_{\mathsf{trigger}}$ is some kind of memory load. Interestingly, we also found a new variant of Flush+Reload that uses `MOVNTDQ` as $\mathsf{Seq}_{\mathsf{reset}}$. This store instruction with a non-temporal hint also evicts the accessed memory address from the cache [43].

Arguably, in a practical attack, this is not very useful, as writable shared memory is typically not a target for Flush+Reload. However, in the case of transient-execution attacks, where an attacker often uses Flush+Reload as a covert channel to transfer the leaked data from the microarchitectural domain to the architectural domain, this alternative flushing method is indeed useful. In Section 6.1, we show that the MOVNT-based Flush+Reload can increase the leakage from 3 to 7.83 bytes



(a) RDRAND (b) XSAVE (c) MMX (d) AVX2 (e) AVX2-x87-FPU

Figure 4: Histograms of $\text{Seq}_{\text{measure}}$ execution time depending on whether $\text{Seq}_{\text{trigger}}$ was executed (solid blue) or not (dashed red).

per transient window for Meltdown-type attacks, reducing the impact of the Flush+Reload part that is often the bottleneck.

AVX2-Based Side Channel. Osiris also found 514 instances of the AVX-based side channel [84]. For this side channel, the $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$ contain AVX2 or AVX512 instructions, and $\text{Seq}_{\text{reset}}$ is simply idling. According to Schwarz et al. [84], a busy-wait executing for around 2 700 000 cycles would power down the AVX2 SIMD unit. However, our manual tests showed that a busy wait of 8000 cycles is, in fact, sufficient.

Interestingly, we also observed during the manual inspection a variant of the AVX2 side channel that contains the PAUSE in its $\text{Seq}_{\text{reset}}$. Figure 4d visualizes the behavior of this new variant for 200 000 executions. As shown in the figure, this variant is, in fact, more stable than the variant based on busy wait. In particular, we observed a difference of 226 cycles between the medians of the two distributions, which is twice the difference for triples that have a busy-wait as $\text{Seq}_{\text{reset}}$.

5.5 Finding Novel Side Channels

To demonstrate the effectiveness of our fuzzer, we tested its ability to uncover new side channels. After running our fuzzer for 21 days, we automatically uncovered 4 different, previously unknown side channels. Table 3 shows an overview of the reported side channels. In the following, we briefly present each of these side channels.

RDRAND-Based Side Channel. This side channel consists of triples having the RDRAND instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and the sleep pseudo-instruction in $\text{Seq}_{\text{reset}}$. Figure 4a visualizes the behavior of this side channel for 200 000 executions. We observed a difference of 228 cycles between the medians of the two distributions. Setting a simple threshold to the average of these two medians leads to a success rate of 84.28 % when attempting to distinguish between the two states S_0 and S_1. While it is unlikely that detecting the execution of the RDRAND instruction leads to a side-channel attack, we demonstrate in Section 6.3 that this finding can be used for a stealthy cross-core covert channel.

XSAVE-Based Side Channel. This side channel consists of triples having the XSAVE or XSAVE64 instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$. For this side channel, $\text{Seq}_{\text{reset}}$ can contain various instructions. However, we distinguish between two variants: (1) a non-transient variant that contains LSL, RDRAND, LAR, FLD, FXRSTOR64, or FXSAVE64 instructions in $\text{Seq}_{\text{reset}}$; and (2) a transient variant that contains XSAVEOPT instruction in addition to most x87-FPU instructions.

Figure 4b visualizes the behaviour for 200 000 executions of a triple formed of XSAVE [R8] in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and LAR ECX, EDX in $\text{Seq}_{\text{reset}}$. We observed a difference of 158 cycles between the medians of the two distributions. Using the average of the two medians as threshold leads to a rather unstable behaviour, though. We observe a success rate of only 75.10 % when attempting to distinguish between the two states S_0 and S_1.

MMX Combined with x87-FPU. This side channel consists of triples having the MMX instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and x87-FPU in $\text{Seq}_{\text{reset}}$. Figure 4c shows the histogram for 200 000 executions of the triples. The reported triples have a time measurement difference of 90 cycles in the median. We could reliably distinguish between the states S_0 and S_1 with an accuracy of 99.99 %.

AVX2 Combined with x87-FPU. This side channel consists of triples having the AVX, AVX2, AVX512, FMA, or F16C instructions in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and x87-FPU in $\text{Seq}_{\text{reset}}$. The reported triples have a time measurement difference in the interval of 72 to 208 cycles.

Figure 4d visualizes the behavior for 200 000 executions of a triple formed of VFMADD132PD YMM1, YMM2, [R8] in both $\text{Seq}_{\text{trigger}}$ and $\text{Seq}_{\text{measure}}$, and FISTP [R8] in $\text{Seq}_{\text{reset}}$. We observe a difference of 166 cycles between the medians of the two distributions. A threshold can distinguish the two states S_0 and S_1 at a success rate of 99.95 %. In Section 6.1, we show that this side-channel leakage can be used for a fast covert channel for Spectre attacks.

6 Case Studies

In this section, we present three case studies based on the newly detected side channels (cf. Section 5). Section 6.1 demonstrates that the newly discovered side channels can be used for transient-execution attacks. They can be used in Spectre attacks to increase the space of possible gadgets, as well as in Meltdown-type attacks to increase the leakage. Section 6.2 introduces a novel microarchitectural attack against

Table 3: Overview of the novel side channels.

Side Channel Name	Example $\text{Seq}_{\text{trigger}}$	Example $\text{Seq}_{\text{measure}}$	Example $\text{Seq}_{\text{reset}}$	Timing Diff.
RDRAND	`RDRAND`	`RDRAND`	Sleep Pseudo-Inst.	228 cycles
XSAVE	`XSAVE [R8]`	`XSAVE [R8]`	`LAR ECX, EDX`	158 cycles
MMX-x87-FPU	`PHADDD MM1, [R8]`	`PHADDD MM1, [R8]`	`FLDLN2`	90 cycles
AVX2-x87-FPU	`VDMADD132PD YMM1, YMM2, [R8]`	`VFMADD132PD YMM1, YMM2, [R8]`	`FISTP [R8]`	166 cycles

kernel-level ASLR (KASLR) based on the results discovered by Osiris. This novel KASLR break even works on the newest Intel Ice Lake and Comet Lake microarchitectures, even if all known mitigations are in place. Section 6.3 shows that the `RDRAND`-based side channel can be used as a cross-core covert channel in the cloud without relying on the cache.

6.1 Transient-Execution Covert Channels

Transient-execution attacks [17], *i.e.*, Spectre- and Meltdown-type attacks, always require a microarchitectural covert channel to transfer the microarchitecturally-encoded data into the architectural state. Typically, these attacks rely on a cache covert channel [17], as also shown in the original Spectre [50] and Meltdown [57] paper. Cache-based covert channels have the advantage that they are ubiquitous, fast, and reliable [17, 50, 57]. In this case study, we show that our new side channels can potentially increase the number of Spectre gadgets, and optimize the leakage for Meltdown-type attacks.

Spectre Attacks. Bhattacharyya et al. [12] and Schwarz et al. [80, 84] already showed different covert channels for Spectre. Their covert channels are based on port contention, vector instructions, and the TLB, respectively. In this case study, we show that our newly discovered side channel based on `AVX2` and `x87-FPU` can also be used for Spectre attacks.

We implement a proof-of-concept Spectre attack that uses this side channel as the covert channel. Our proof of concept exploits Spectre-PHT [50] to leak a string outside of the bounds of an array. We can use the same gadgets as in a NetSpectre attack [84] and similar gadgets as used in SMoTherSpectre [12]. More specifically, exploiting the discovered side channels would require finding specific gadgets (conditional trigger sequence) in the victim code. Such gadgets could also be constructed in combination with other Spectre vulnerabilities using speculative ROP [11, 12]. Depending on the value of a transiently accessed bit, an `AVX2` instruction is executed or not executed. While NetSpectre simply waits for the state to be reset, we rely on the findings of Osiris that executing an `x87-FPU` instruction resets the state faster. The receiving end of the covert channel is again an `AVX2` instruction. We tested our code on an Intel Core i7-9700K, where we achieved a leakage rate of 2407 bit/s with an error rate of 0.43 %. This is 2.4 times as fast as the transmission rate of the `AVX`-based covert channel used in NetSpectre [84].

Meltdown Attacks. In Meltdown-type attacks, both the sending and the receiving end of the covert channel are entirely attacker-controlled. So far, all Meltdown-type attacks [15, 17, 57, 77, 82, 87, 91, 93] relied on the cache and typically on Flush+Reload to recover the information from the cache. Even though Flush+Reload is extremely fast and reliable, it is still the bottleneck for leaking data [57].

With Stream+Reload, we introduce a new cache attack for improving the leakage rate of Meltdown-type attacks. Stream+Reload is based on the discovery of Osiris that non-temporal memory stores flush the target from the cache. While a cache attack that requires shared writable memory is not useful in a typical side-channel scenario, it is ideal as a fast covert channel for transient-execution attacks. Stream+Reload replaces the `CLFLUSH` instruction with a `MOVNTDQ` instruction. The `MOVNTDQ` instruction has a similar effect as the `CLFLUSH` instruction. It evicts the target cache line from the cache [43].

Reliability of Eviction. Using L3 performance counters, we confirmed that the `MOVNTDQ` instruction indeed reliably evicts the cache line from all cache levels. With respect to the eviction reliability, there is no difference between `MOVNTDQ` and `CLFLUSH` or `CLFLUSHOPT`. Both for Stream+Reload and Flush+Reload, we measured an F-score of 1.0 (n = 10 000 000). Furthermore, even novel cache designs [59, 76, 97] likely do not prevent this type of eviction, as they only block the flush instruction and prevent the efficient creation of eviction sets.

Performance. We observe one significant difference between Flush+Reload and Stream+Reload. Although in both attacks, the value is evicted from all cache levels, the reload of a value flushed using `MOVNTDQ` is significantly faster on all our tested CPUs. On the i7-8565U, for example, reloading a value when it was flushed takes on average 253 cycles (n = 20 000 000) (including an `MFENCE` each before and after the memory load). In contrast, when the value was evicted using `MOVNTDQ`, reloading only takes 172 cycles (n = 20 000 000). Analyzing the uncore performance counters shows that this time difference for loading the data originates from the uncore (`offcore_requests_outstanding.cycles_with_data_rd`). We attribute the time difference to the cache-coherency protocol. Flushing the cache line puts the cache line into the *invalid* state, while writing to the cache line puts it into the *modified* state [66, 71]. When loading the flushed cache line, it switches to the *exclusive* state, while the *modified* state stays the same. Due to the different behaviors of cache snooping, loading from different cache coherence states also results in different latencies [66].

Results. The faster reload time allows encoding 2.5x more values during the transient window. In a Meltdown proof of concept relying on Stream+Reload, we can, on average, leak 7.83 bytes at once ($n = 100\,000$) (Intel i3-5010U).[4] Previous work was only able to leak up to 3 bytes [57, 65, 77, 82].

6.2 MOVNT-based KASLR Break

KASLR has been subject to almost countless microarchitectural attacks in the past [15, 16, 24, 33, 42, 49, 62, 80]. As a response, researchers, CPU vendors, and OS maintainers have developed several countermeasures [2, 16, 29, 32]. In particular, the newest 10th-generation Intel CPUs (Ice Lake and Comet Lake) are immune to many microarchitectural KASLR breaks, including the recently discovered EchoLoad attack [16]. However, our newly-discovered side channel can be used to break KASLR even on those architectures.

Based on the discovery of Osiris that the `MOVNT` instruction evicts a cache line, we manually evaluated whether this eviction also works for inaccessible addresses such as kernel addresses. Previous work showed that even for Meltdown-resistant CPUs, memory loads [16, 92] and stores [80] can infer side-channel information from the kernel. Although `MOVNT` could not directly evict kernel memory, we observed changes in the cache state on seemingly unrelated memory. If the targeted kernel address is invalid, *i.e.*, not physically backed, we observe that an unrelated `MOV` on user memory issued after the `MOVNT` fails. If the kernel address is physically backed, the `MOV` is successful. Hence, this allows de-randomizing the location of the kernel, effectively breaking KASLR.

```
1  try {
2    asm volatile(
3      "clflush 0(%[probe])\n"
4      "movq %%rsi, (%[dummy])\n"
5      "movntdqa (%[kernel]), %%xmm1\n"
6      "movq (%[probe]), %%rax\n"
7    ) : : [probe]"r"(probe), [dummy]"r"(dummy),
8          [kernel]"r"(kernel)
9      : "rax", "xmm1", "rsi", "memory");
10 } catch {
11   if(uncached(probe)) return MAPPED;
12   else return UNMAPPED;
13 }
```

Listing 1: The main part of FlushConflict. The probe memory is uncached if the kernel address is physically backed.

Listing 1 shows the minimal working example of our KASLR break, FlushConflict, that we created from our findings on `MOVNT`. A user-accessible memory address (`probe`) is flushed, followed by a write to an unrelated address, acting as a reordering barrier. Afterward, the kernel address (`kernel`) is read using `MOVNT`. Finally, `probe` is accessed. As the load from the kernel address leads to a fault, exceptions are handled using a signal handler for this code. After resolving the fault, the cache state of `probe` is observed, e.g., using Flush+Reload. If `probe` is cached, the kernel address is invalid, if `probe` is not cached, the kernel address is valid.

[4] We used this older CPU as the new CPUs are not affected by Meltdown.

Table 4: The evaluated CPUs for the KASLR break.

CPU (Microarchitecture)	Accuracy (idle)	Accuracy (stress)	Runtime
Intel Core i5-3230M (Ivy Bridge)	99 %	97 %	34 ms
Intel Core i5-4690 (Haswell)	100 %	99 %	221 ms
Intel Core i3-5010U (Broadwell)	99 %	97 %	5 ms
Intel Core i7-6700K (Skylake)	99 %	98 %	9 ms
Intel Core i7-8565U (Whiskey Lake)	100 %	92 %	6 ms
Intel Core i7-9700K (Coffee Lake)	100 %	98 %	102 ms
Intel Core i9-9980HK (Coffee Lake)	99 %	99 %	65 ms
Intel Core i3-1005G1 (Ice Lake)	96 %	96 %	300 ms
Intel Core i7-10510U (Comet Lake)	99 %	97 %	84 ms
Intel Celeron J4005 (Gemini Lake)	99 %	99 %	349 ms
Intel Xeon Platinum 8124M (Skylake-SP)	99 %	99 %	318 ms

Root-Cause Hypothesis. Using performance counters, we analyzed the behavior of FlushConflict. The `CLFLUSH` and load access to the same address trigger a cache-line conflict as also exploited in ZombieLoad [82]. Even though, at first, the write to `dummy` seems unrelated, it is guaranteed to be ordered with `CLFLUSH` [45] and hence influences the overall timing of the executed code in the processor pipeline. Alternatively, this line can also be removed entirely (depending on the CPU) or replaced by a different method to add a delay, e.g., using a dummy loop. However, adding a serializing instruction, such as a fence, breaks the attack, as it forces the `CLFLUSH` to retire, preventing the cache-line conflict with the load. If `kernel` is physically backed, we observe a page-table walk (`dtlb_load_misses.miss_causes_a_walk`). If `kernel` is *not* physically backed, we observe 2 page-table walks, *i.e.*, the page-table walk is repeated. That is in agreement with Canella et al. [16], showing that loads from non-present kernel pages are re-issued. As this case takes longer [49] and faults are only detected at the retirement of instructions, it gives other out-of-order executed instructions more time to execute. We hypothesize that if the kernel address is unmapped, the processor has a long-enough speculation window to execute the flush, write, and the last load. As a result of this, the last load brings `probe` back to the cache. In the case of a mapped kernel address, the processor detects the fault earlier and hence stops the execution before the last load was issued. As a result, `probe` is cached if `kernel` is not physically backed, and not cached if `kernel` is physically backed. The observed performance counters back this hypothesis. For an unmapped address, `mem_load_retired_l3_miss` shows fewer events. However, the number of cycles spent waiting for memory (`cycle_activity.cycles_l3_miss`) is slightly higher. This indicates that there are ongoing load instructions that never retire, backing the hypothesis that the last load is only executed transiently when the address is unmapped.

Applicability. We tested our microarchitectural KASLR break on Intel CPUs from the 3^{rd} to the 10^{th} generation, *i.e.*,

Table 5: The evaluated CPUs for the RDRAND covert channel.

CPU	Setup	Cross-HT		Cross-Core	
		Speed	Error	Speed	Error
Intel Core i5-3230M	Lab	133.3 bit/s	8.87 %	133.3 bit/s	0.05 %
Intel Core i3-5010U	Lab	666.7 bit/s	0.30 %	333.3 bit/s	1.82 %
Intel Core i7-8565U	Lab	400.0 bit/s	0.65 %	166.7 bit/s	0.63 %
Intel Core i9-9980HK	Lab	500.0 bit/s	0.76 %	117.6 bit/s	9.25 %
Intel Core i3-1005G1	Lab	1000.0 bit/s	0.37 %	1000.0 bit/s	0.00 %
Intel Xeon E5-2686 v4	Cloud	500.0 bit/s	0.21 %	333.3 bit/s	2.48 %
Intel Xeon E5-2666 v3	Cloud	666.7 bit/s	2.64 %	95.2 bit/s	0.88 %
AMD Ryzen 5 2500U	Lab	48.8 bit/s	2.80 %	48.8 bit/s	2.00 %
AMD Ryzen 5 3550H	Lab	666.7 bit/s	2.10 %	500.0 bit/s	2.50 %

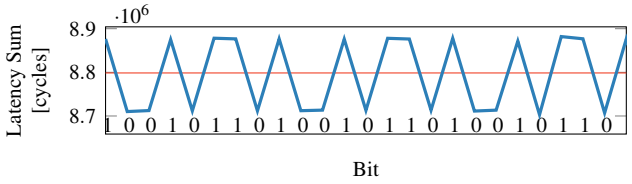


Figure 5: Using the RDRAND covert channel to send the bit stream 100101101001011010010110... from one CPU core to a different physical core (Intel Core i3-1005G1).

from Ivy Bridge to Comet Lake. As shown in Table 4, we used desktop (Core), server (Xeon), and mobile (Celeron) CPUs.

In contrast, we experimentally verified that EchoLoad [16], which works on a large range of Intel CPUs from 2010 to 2019, does not work on Ice Lake or Comet Lake. We confirm that the KASLR break is operating-system agnostic by successfully mounting it on Linux and Windows 10.

In the case of KPTI, *i.e.*, on CPUs that are not Meltdown-resistant, the KASLR break detects the trampoline used to switch to the kernel. Otherwise, if the CPU is Meltdown-resistant or KPTI is disabled, the KASLR break detects the start of the kernel image. As an unprivileged attacker can read out the state of KPTI and whether the CPU is vulnerable to Meltdown, the attacker always knows the start of the kernel image. Moreover, as the kernel image itself is not randomized, knowing the kernel version and the start of the kernel image is sufficient to calculate the location of any kernel part.

Additionally, we tested the KASLR break by simulating a realistic environment by artificially raising the pressure on the CPU and memory subsystem using the *stress* utility. We still observe success rates ranging from 92% to 99% for the different microarchitectures ($n = 100$). Furthermore, we verified the KASLR break in a cloud scenario by testing it on an Intel Xeon Platinum 8124M in the AWS cloud.

Performance. On average, our KASLR break detects the start of the kernel image within 136 ms ($n = 1100$) While not the fastest microarchitectural KASLR break, it is on par with other microarchitectural KASLR breaks [16].

6.3 RDRAND Covert Channel in the Cloud

Osiris discovered a timing leakage in the RDRAND instruction on both Intel and AMD CPUs. In this section, we present a cross-core covert channel based on these timing differences. We evaluate the capacity in a cross-thread scenario (Section 6.3.2), and across cores and VMs (Section 6.3.3). Finally, we analyze the leakage reason (Section 6.3.4).

6.3.1 Setup

The setup consists of a sender and a receiver application. In our proof-of-concept implementation, sender and receiver are simply time-synchronized, *i.e.*, they rely on a common time source such as the timestamp counter. To send a '1'-bit, the sender repeatedly executes the RDRAND instruction for a fixed time τ. To send a '0'-bit, the sender idles for τ. The receiver measures the latency of the RDRAND instruction over a period of τ. The latency directly corresponds to the sent bit, *i.e.*, a high latency is caused by a '1'-bit, and a low latency is caused by a '0'-bit. We note that this setup is not optimal, as there are more advanced techniques for synchronization, including error correction [22, 64, 99]. However, our goal is to show the feasibility and the noise-resistance of this channel, not how far it can be optimized using better engineering.

6.3.2 Same-core Leakage

We evaluated an RDRAND-based covert channel across hyperthreads to estimate the maximum capacity of this channel. Note that the leakage in a cross-hyperthread channel is boosted by port contention as well [7, 12]. Moreover, on Intel CPUs, Intel documents that the microcode update preventing SRBDS [77] serializes RDRAND executions on the same core [47]. Hence, to rule out any influence of the microcode fixes, we evaluated the channel with and without the active patches. As AMD CPUs are not susceptible to SRBDS, there is no microcode influence to rule out. As Table 5 shows, we verified the covert channel on all Intel microarchitectures since at least the Ivy Bridge microarchitecture, and also on the AMD Zen and Zen+ microarchitecture. We achieve the best results on the newest microarchitectures, with 1000 bit/s (0 % error) on Intel and 666.7 bit/s (2.1 % error) on AMD. While a same-core channel is usually irrelevant, it shows the upper bound of the leakage achievable across cores.

6.3.3 Cross-core Leakage

In addition to the expected leakage across hyperthreads, we evaluate the channel across physical cores.

Local Environment. Figure 5 shows a cross-core transmission in a local environment. While the signal is weaker than in the cross-hyperthread scenario, we still manage to transmit data reliably. As shown in Table 5, the channel achieves up to 1000 bit/s with a low error rate down to 0 %.

AWS Cloud. To further evaluate the applicability of the covert channel in a real-world scenario, we mounted it be-

Table 6: Transmission and error rates of state-of-the-art cross-core covert channels sorted by transmission speed.

Covert channel (Element)	Speed	Error rate
Liu et al. [60] (L3)	600 kbit/s	1.00 %
Pessl et al. [75] (DRAM)	411 kbit/s	4.11 %
Maurice et al. [64] (L3)	362 kbit/s	0.00 %
Evtyushkin et al. [22] (RDSEED)	71 kbit/s	0.00 %
Ragab et al. [77] (CPUID)	24 kbit/s	5.00 %
Ours (RDRAND)	1000 bit/s	0.00 %
Maurice et al. [63] (L3)	751 bit/s	5.70 %
Wu et al. [99] (memory bus)	747 bit/s	0.09 %
Semal et al. [85] (memory bus)	480 bit/s	5.46 %
Schwarz et al. [83] (DRAM)	11 bit/s	0.00 %

tween two virtual machines running in the AWS cloud. To ensure that we do not interfere with other users, we used a dedicated C3 host with an Intel Xeon E5-2666 v3. We were able to transmit 95.2 bit/s across two different virtual machines running on the same CPU with an error rate of 0.88 %. Additionally, the host had a third virtual machine running to simulate realistic noise. For completeness, we also verified that the covert channel works across hyperthreads and cores inside a single virtual machine in this setup (cf. Table 5).

Comparison to Other Cross-Core Covert Channels. Table 6 shows a comparison of the transmission speed for state-of-the-art cross-core covert channels. While the `RDRAND`-based covert channel is much slower than modern cache-based covert channels, it has two huge advantages. First, there are no performance counters for the hardware random number generator. Thus, this channel cannot be easily detected or prevented by current approaches relying on performance counters [19, 40, 48, 72]. We also used the open-source HexPADS framework [72] to verify that it cannot detect the covert channel. Second, in contrast to memory-based covert channels, this channel is agnostic to any typical system noise caused by memory accesses on the sender core. As typical workloads do not execute `RDRAND` in a high frequency, we do not see a high impact on the transmission rate, even for high workloads. We verified that by running the Linux tool `stress` for both the CPU and the memory on the sender core does not prevent the covert channel. Even in this scenario, with an extremely high load of 100 % on the sibling hyperthread, we manage to transmit 500.0 bit/s with an error rate of 7.34 %.

Furthermore, as our covert channel does not rely on the memory subsystem, defenses proposed against cache attacks [59, 76, 96, 97, 104, 105] do not prevent our channel. Even existing partitioning features, such as Intel CAT, which can be used to prevent cache-based cross-VM covert channels [58] do not affect the `RDRAND`-based covert channel.

6.3.4 Explanation for RDRAND Side Channel

As the hardware random number generator is shared across all cores, simultaneous use by multiple cores leads to contention. Hence, as with many cross-core covert channels [22, 63, 75, 99], the root cause is the contention of a resource shared across cores, such as the L3 cache or the memory bus. However, in contrast to previous covert channels, we could not identify any performance counters related to `RDRAND`. While this makes the analysis more difficult, it also increases the stealthiness of the channel, as it cannot be detected easily.

While previous work showed that the `RDSEED` instruction can exhaust the hardware random-number generator (RNG) [22], the `RDRAND` instruction has not been analyzed for side-channel leakage. Moreover, Evtyushkin et al. [22] only exploited an architectural value, *i.e.*, a cleared carry flag, indicating that the RNG is exhausted, and not differences in the execution time. At first glance, it might seem obvious that `RDRAND` also suffers from exhaustion as it fundamentally relies on the `RDSEED` instruction. `RDSEED` is quickly exhausted, as it provides the randomness directly from the hardware element. However, Evtyushkin et al. [22] observed that `RDRAND` provides the numbers from a pseudo-RNG and can thus provide continuous streams of numbers. We confirm that the `RDRAND`-based leakage is *not* due to exhaustion. While measuring the timing differences, the instruction does not indicate that the RNG is exhausted, *i.e.*, the carry flag was always set [44].

We additionally ruled out the microcode updates preventing CrossTalk [77] as a cause for the timing differences. While these updates reduce the bandwidth of `RDRAND` across hyperthreads due to serialization, they do not affect the cross-core behavior [47]. We verified that by successfully mounting the covert channel with and without the microcode update, and also by disabling the mitigation on patched systems via the `IA32_MCU_OPT_CTRL` model-specific register.

7 Discussion

With Osiris, we present a generic approach for detecting timing-based side channels. Our current prototype still has several limitations preventing it from finding even more side channels. However, these are not conceptual limitations. It would merely require a lot more engineering to solve them. In the current version, we only consider side channels where the timing difference is around 100 cycles. Any side channel with a smaller timing difference, e.g., Flush+Flush [35], CacheBleed [102] or the AMD way predictor [56], is currently not reported. One practical reason is that Osiris runs on a commodity Linux system, where it is tough to eliminate all influences on the measurement. Even when isolating cores, several microarchitectural elements are shared across all cores, there are still remaining interrupts, and the power management of the CPU can change the CPU frequency, e.g., for thermal reasons. Hence, to reliably detect small timing

differences, Osiris would have to run on a custom operating system designed for microarchitectural research, such as Sushi Roll [26]. In line with related work [27, 30], our prototype only considers sequences consisting of one instruction. As a consequence, eviction-based side channels such as Evict+Reload, Evict+Time, Prime+Probe, or Reload+Refresh are not detected. However, related work [34, 94, 95] showed that eviction strategies can also be found automatically. Moreover, for specific problems, the search space can be reduced by mutating existing instruction sequences (similar to Medusa [65]) or instruction operands instead of randomly generating them. Therefore, Osiris can be augmented by these techniques to also find eviction-based side channels and support multi-instruction sequences (e.g., fault suppression). Furthermore, using performance counters, power (RAPL), and debug interfaces (Intel VISA/ITP-XDP) as feedback mechanisms, the fuzzer could monitor resource usage and microarchitectural conflicts to guide the sequence generation process. This would allow finding eviction-based channels: (i) Start with multiple loads as a reset sequence, (ii) Mutate the loaded addresses while maximizing (guidance) the cache miss count until a time difference is detected.

Still, despite these current limitations of the prototype, Osiris discovered novel timing-based side channels within hours of runtime. These side channels led to the discovery of a new microarchitectural KASLR break, a previously unknown cross-VM covert channel, and an improvement for transient-execution attacks. Hence, we argue that Osiris is a useful tool for automating the search for timing-based side channels that can also be used by CPU vendors to detect such side channels introduced by new ISA extensions automatically.

Also, Osiris can be extended to other architectures, e.g., ARMv8, with relative ease. To this end, the main parts that need to be adapted are the code generation stage, particularly the offline phase to construct possible instruction variants, and the execution stage. The current implementation of Osiris uses inlined instructions to measure the execution time, which would need to be changed for the target architecture (see Section 4). However, this task can be simplified by refining the current approach to use other timing primitives [55].

8 Conclusion

Our findings illustrate that prior side channels targeted only a subset of many micro-architectural changes. We show several additional, undocumented instruction side effects that attackers can leverage for security-critical side channels. This has severe implications to existing and future side-channel defenses, as each of them is based on a specific threat model that frames (known) attack capabilities. We, therefore, see our proposed fuzzing-based technique as the first *systematic*, *generic*, and *automated* attempt to fast-forward the arms race of detecting (and then, ultimately, defending against) such side channels. The newly discovered side channels and their application to three use cases raise our confidence that Osiris can indeed support this endeavor. When used during the CPU design stage, Osiris helps to eliminate—or at least to document—side channels early on. For this reason, we released Osiris as an open-source tool.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mathias Payer, for their helpful comments and suggestions that substantially helped in improving the paper, as well as Moritz Lipp (Graz University of Technology) for feedback on an earlier version of the paper. Furthermore, we thank the Saarbrücken Graduate School of Computer Science for their funding and support for Daniel Weber. This work partially was supported by grant from the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ:13N1S0762).

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *ASPLOS*, 2019.

[2] Accardi, Kristen Carlson. Function Granular KASLR, 2020. URL: https://patchwork.kernel.org/project/kernel-hardening/list/?series=354389.

[3] Onur Acıiçmez, Shay Gueron, and Jean-pierre Seifert. New Branch Prediction Vulnerabilities in OpenSSL and Necessary Software Countermeasures. In *Proceedings of the 11th IMA International Conference on Cryptography and Coding*, 2007.

[4] Onur Acıiçmez, Çetin Kaya Koç, and Jean-pierre Seifert. On the Power of Simple Branch Prediction Analysis. In *AsiaCCS*, 2007.

[5] Onur Acıiçmez and Werner Schindler. A Vulnerability in RSA Implementations Due to Instruction Cache Analysis and Its Demonstration on OpenSSL. In *CT-RSA 2008*. 2008.

[6] Onur Acıiçmez, Jean-Pierre Seifert, and Çetin Kaya Koç. Predicting secret keys via branch prediction. In *CT-RSA*, 2007.

[7] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2018.

[8] Arm. A-Profile Exploration tools, 2017. URL: https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools.

[9] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *NDSS*, 2019.

[10] Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Hardware prefetchers leak: A revisit of SVF for cache-timing attacks. In *MICRO*, 2012.

[11] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. Specrop: Speculative exploitation of ROP chains. In *RAID*, San Sebastian, 2020.

[12] 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.

[13] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[14] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *USENIX Security Symposium*, 2020.

[15] 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.

[16] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[17] 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/.

[18] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.

[19] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using hardware performance counters. ePrint 2015/1034, 2015.

[20] Christopher Domas. Breaking the x86 ISA, v. 2017-07-27. *Black Hat US*, 2017.

[21] Michael Eddington. Peach Fuzzer. URL: https://www.peach.tech/.

[22] Dmitry Evtyushkin and Dmitry Ponomarev. Covert channels through random number generator: Mechanisms, capacity estimation and mitigations. In *CCS*, 2016.

[23] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Covert channels through branch predictors: a feasibility study. In *HASP*, 2015.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR. In *MICRO*, 2016.

[25] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[26] Brandon Falk. Sushi Roll: A CPU research kernel with minimal noise for cycle-by-cycle microarchitectural introspection, 2019. URL: https://gamozolabs.github.io/metrology/2019/08/19/sushi_roll.html.

[27] Anders Fogh. Covert Shotgun: automatically finding SMT covert channels, 2016. URL: https://cyber.wtf/2016/09/27/covert-shotgun/.

[28] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2016.

[29] David Gens, Orlando Arias, Dean Sullivan, Christopher Liebchen, Yier Jin, and Ahmad-Reza Sadeghi. LAZARUS: Practical Side-Channel Resilient Kernel-Space Randomization. In *RAID*, 2017.

[30] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*, 2020.

[31] Daniel Gruss. *Software-based Microarchitectural Attacks*. PhD thesis, Graz University of Technology, 2017.

[32] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[33] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS*, 2016.

[34] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, 2016.

[35] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[36] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*, 2015.

[37] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*, 2011.

[38] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *NDSS*, 2019.

[39] Aki Helin. Radamsa. URL: https://gitlab.com/akihe/radamsa.

[40] Nishad Herath and Anders Fogh. These are Not Your Grand Daddys CPU Performance Counters – CPU Hardware Performance Counters for Security. In *Black Hat Briefings*, 2015.

[41] Sam Hocevar. Zzuf. URL: https://github.com/samhocevar/zzuf/.

[42] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P*, 2013.

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

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

[45] Intel. Intel 64 and IA-32 Architectures Software Developer′s Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[46] Intel. Affected Processors: Transient Execution Attacks, 2020. URL: https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.

[47] Intel. Special Register Buffer Data Sampling, 2020. URL: https://software.intel.com/security-software-guidance/deep-dives/deep-dive-special-register-buffer-data-sampling.

[48] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Mascat: Preventing microarchitectural attacks before distribution. In *CODASPY*, 2018.

[49] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS*, 2016.

[50] 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.

[51] Paul C. Kocher. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*, 1996.

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

[53] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *S&P*, 2020.

[54] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[55] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[56] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*, 2020.

[57] 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.

[58] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst:

Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[59] Fangfei Liu and Ruby B. Lee. Random Fill Cache Architecture. In *MICRO*, 2014.

[60] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

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

[62] Giorgi Maisuradze and Christian Rossow. Speculose: Analyzing the Security Implications of Speculative Execution in CPUs. *arXiv:1801.04084*, 2018.

[63] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. C5: Cross-Cores Cache Covert Channel. In *DIMVA*, 2015.

[64] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *NDSS*, 2017.

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

[66] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Wolfgang E Nagel. Cache Coherence Protocol and Memory Performance of the Intel Haswell-EP Architecture. In *ICPP*, 2015.

[67] John Monaco. SoK: Keylogging Side Channels. In *S&P*, 2018.

[68] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, 2015.

[69] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*, 2006.

[70] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Zest: Validity fuzzing and parametric generators for effective random testing. *arXiv:1812.00078*, 2018.

[71] Salvador Palanca, Stephen A Fischer, and Subramaniam Maiyuran. CLFLUSH micro-architectural implementation method and system, 2003. US Patent 6,546,462.

[72] Matthias Payer. HexPADS: a platform to detect "stealth" attacks. In *ESSoS*, 2016.

[73] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *IEEE S&P*, 2018.

[74] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[75] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[76] Moinuddin K Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *IEEE MICRO*, 2018.

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

[78] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, 2017.

[79] Michael Schwarz. *Software-based Side-Channel Attacks and Defenses in Restricted Environments*. PhD thesis, Graz University of Technology, 2019.

[80] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725*, 2019.

[81] Michael Schwarz, Moritz Lipp, Daniel Gruss, Samuel Weiser, Clémentine Maurice, Raphael Spreitzer, and Stefan Mangard. KeyDrown: Eliminating Software-Based Keystroke Timing Side-Channel Attacks. In *NDSS*, 2018.

[82] 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.

[83] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *FC*, 2017.

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

[85] Benjamin Semal, Konstantinos Markantonakis, Keith Mayes, and Jan Kalbantner. One covert channel to rule them all: A practical approach to data exfiltration in the cloud. In *TrustCom*, 2020.

[86] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage. In *CCS*, 2018.

[87] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480*, 2018.

[88] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *NDSS*, 2016.

[89] Dean Sullivan, Orlando Arias, Travis Meade, and Yier Jin. Microarchitectural Minefields: 4K-aliasing Covert Channel and Multi-tenant Detection in IaaS Clouds. In *NDSS*, 2018.

[90] M Caner Tol, Koray Yurtseven, Berk Gulmezoglu, and Berk Sunar. Fastspec: Scalable generation and detection of spectre gadgets using neural embeddings. *arXiv:2006.14147*, 2020.

[91] 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.

[92] 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.

[93] 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.

[94] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. CacheQuery: Learning Replacement Policies from Hardware Caches. In *PLDI*, 2020.

[95] Pepe Vila, Boris Köpf, and Jose Morales. Theory and Practice of Finding Eviction Sets. In *S&P*, 2019.

[96] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *ACM SIGARCH Computer Architecture News*, 35(2), 2007.

[97] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security Symposium*, 2019.

[98] Henry Wong. The microarchitecture behind meltdown, may 2018. URL: http://blog.stuffedcow.net/2018/05/meltdown-microarchitecture/.

[99] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *USENIX Security Symposium*, 2012.

[100] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

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

[102] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA. *JCEN*, 2017.

[103] Michal Zalewski. Technical "whitepaper" for afl-fuzz, 2014. URL: http://lcamtuf.coredump.cx/afl/technical_details.txt.

[104] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency Detection in the Cloud via Side-Channel Analysis. In *S&P*, 2011.

[105] Yinqian Zhang and MK Reiter. Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.

A Clustering Results

Table 7 shows the clustering results for the CPUs on which Osiris ran. Osiris found multiple thousand side channels that were clustered based on the instruction extension of $\mathsf{Seq}_{\mathsf{trigger}}$, $\mathsf{Seq}_{\mathsf{measure}}$, and $\mathsf{Seq}_{\mathsf{reset}}$, resulting in 100 to 200 clusters. However, as $\mathsf{Seq}_{\mathsf{reset}}$ is typically not involved in the actual leakage, clustering based on the instruction extension of only $\mathsf{Seq}_{\mathsf{trigger}}$ and $\mathsf{Seq}_{\mathsf{measure}}$ results in a smaller number of clusters.

Table 7: Cluster Results For Intel Microarchitectures.

CPU Name	Found	Extension	$\mathsf{Seq}_{\mathsf{measure}}$-$\mathsf{Seq}_{\mathsf{trigger}}$ only
Intel Core i7-9750H	68 597	186 clusters	16 clusters
Intel Core i5-4690	51 468	168 clusters	19 clusters
Intel Core i7-9700K	27 512	104 clusters	26 clusters