Shesha: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors

Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay,
Indian Institute of Technology Kharagpur

https://www.usenix.org/conference/usenixsecurity24/presentation/chakraborty

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Shesha*: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors

Anirban Chakraborty
Indian Institute of Technology Kharagpur
anirban.chakraborty@iitkgp.ac.in

Nimish Mishra
Indian Institute of Technology Kharagpur
nimish.mishra@kgpian.iitkgp.ac.in

Debdeep Mukhopadhyay
Indian Institute of Technology Kharagpur
debdeep@cse.iitkgp.ac.in

Abstract

Transient execution attacks have been one of the widely explored microarchitectural side channels since the discovery of Spectre and Meltdown. However, much of the research has been driven by manual discovery of new transient paths through well-known speculative events. Although a few attempts exist in literature on automating transient leakage discovery, such tools focus on finding variants of known transient attacks and explore a small subset of instruction set. Further, they take a random fuzzing approach that does not scale as the complexity of search space increases. In this work, we identify that the search space of bad speculation is disjointedly fragmented into *equivalence classes*, and then use this observation to develop a framework named Shesha, inspired by Particle Swarm Optimization, which exhibits faster convergence rates than state-of-the-art fuzzing techniques for automatic discovery of transient execution attacks. We then use Shesha to explore the vast search space of extensions to the `x86` Instruction Set Architecture (ISAs), thereby focusing on previously unexplored avenues of bad speculation. As such, we report five previously unreported transient execution paths in Instruction Set Extensions (ISEs) on new generation of Intel processors. We then perform extensive reverse engineering of each of the transient execution paths and provide root-cause analysis. Using the discovered transient execution paths, we develop attack building blocks to exhibit exploitable transient windows. Finally, we demonstrate data leakage from Fused Multiply-Add instructions through SIMD buffer and extract victim data from various cryptographic implementations.

1 Introduction

With a plethora of sophisticated optimizations incorporated in modern processors, speculative execution plays a crucial role in determining the overall performance of the system. However, in 2018 a new class of attacks, called *transient attacks* introduced with Spectre [30] and Meltdown [33], have brought into perspective the side-channel implication of speculative execution. These attacks rely on software-accessible side-channels to extract secrets by forcing the CPU to enter into *transient state* and leave microarchitectural traces. A number of attacks have been demonstrated in literature that exploit these vulnerabilities to leak secret information across security boundaries [2, 4, 8, 9, 11, 12, 19, 29, 31, 34, 35, 41–44, 46–51, 53, 54, 56]. Intel refers to such cases as *bad speculation* [24] where the CPU is being coerced into a condition where the issued micro-operations (μOps) have to be discarded while their results are transiently reflected in microarchitectural states.

As per Intel, bad speculation happens due to three broader causes - branch misprediction, microcode assists and machine clear [24]. While a number of works have explored branch misprediction-based transient attacks in literature [9, 12, 30, 31, 34], very little research has been made to explore machine clears and microcode assists. Recently, authors in [40] explored the occurrences of machine clears in modern processors and demonstrated multiple transient execution paths based on different machine clear events. The findings of [40] open up a new frontier for transient execution discoveries that motivate further exploration. While most of the known works on transient leakage discoveries have been the results of manual effort, recent literature has seen quite some attempts to automate such leakage exploration [21, 22, 36, 52, 55]. However, these tools were developed to target certain classes of speculations and cannot be directly extended to explore the vast space of bad speculation (refer Section 9 for a detailed discussion and comparison with our tool). Additionally, due to the large number of x86 instructions, the prior works on automation only consider a small subset of Instruction Set Extensions (ISE), thereby missing out on a large pool of specialized instructions that are performance-bound and share various hardware resources. In fact, some of these ISEs, especially those concerned with SIMD, streaming SIMD, and Vector extensions, have their own separate data path (i.e. execution units), registers, temporal buffers, etc, making them an

*Shesha or Sheshanaga is a serpentine demigod in Hindu mythology with multiple heads and serves as the celestial bed of Lord Vishnu.

ideal spot for enabling speculation, and possibly bad speculation. Evidently, a systematic exploration of bad speculation in the context of ISEs is necessary to understand transient execution related vulnerabilities of ISEs. Concretely, in this work, we deal with the following question:

Given the wide-array of ISEs, can we develop a generic, better-than-random (i.e. better than fuzzing) approach to automatically explore the vast search space of bad speculation across ISE executions and develop previously unexplored transient executions paths?

In this work, we choose to adopt an alternative path than random fuzzing for the following reasons:

1. **Undirected vs Directed searches**: Fuzzing without feedback (like coverage) is an undirected approach that cannot incrementally develop upon good solutions since it applies random mutations without any sense of direction. In contrast, directed approaches (like evolutionary algorithms) consistently try to make good solutions *better*. This ensures faster convergence for directed approaches as opposed to undirected fuzzing-based approaches.
2. **Scalability with vastly expressive search spaces**: Randomized, undirected fuzzing leads to scalability issues with vastly expressive search spaces. With modern ISAs (along with extensions), the total number of possible instructions exceeds 16000. Random fuzzing through such expansive search spaces causes problems, as evident in prior works in the literature. For instance, works like [21, 36, 39] limit the extent of their instruction set to prevent blowing up the search space.

Given these observations, we choose to develop a guided approach that will not require restricting the search space, thereby increasing chances of finding corner cases of bad speculation. We develop a framework- Shesha - based on the principles of *Particle Swarm Optimizers (PSO)* [28] to allow for a directed exploration of the ISE search space, and faster convergence on code sequences enabling bad speculation. We choose PSO as the foundation of Shesha since PSOs allow for a delicate balance between choosing from local and global gains in the search space. However, unlike a textbook PSO, we observe that the search space for bad speculation is disjointedly fragmented, and therefore allows modifications to the generic PSO that enable faster convergence. Consequently, Shesha is able to uncover newer, previously unreported avenues of bad speculation in ISEs: (1) SIMD-Vector instruction intermixing (which has been unobserved [40] in newer generation Intel processors prior to Alder Lake and Sapphire Rapids), (2) execution of fused-multiply-add (FMA) instructions, (3) intermixing single and double precision in ISEs, and (4) performing denormal arithmetic on combination of AES-NI and SIMD instruction sets. We perform extensive reverse engineering of each of these avenues to uncover the cause of bad speculation in each of these cases, which helps us to construct abstract code sequence descriptions called *gadgets*, vulnerable to transient execution attacks. To summarize, we make the following contributions:

- We propose Shesha, a framework to explore previously untapped search space of bad speculation with respect to *Instruction Set Extensions*. Our methodology relies on a modified version of Particle Swarm Optimization and exploits the disjointedly fragmented nature of transient execution attacks to deliver faster convergence rates.
- We uncover newer and previously unreported avenues of transient execution paths: SIMD-Vector transitions, execution of fused-multiply-add (FMA) instructions, intermixing single and double precision in ISEs, and performing denormal arithmetic on a combination of AES-NI and SIMD instruction sets. We also provide extensive root-cause analysis for each of the scenarios.
- We characterize and evaluate the leakage strengths of each of the uncovered transient execution paths and discover SIMD-Vector transitions to exhibit the *longest* adversarial-friendly window of execution amongst all transient execution paths. We also demonstrate Simultaneous Multi-Threaded covert channels as well as Load Value Injection (LVI) attacks on vulnerable targets.
- Finally, we reverse-engineer the Fused Multiply-Add (FMA) execution unit to root-cause the transient leakage and exploit it to leak victim data from multiple cryptographic implementations. Our findings suggest that data used by *all* FMA and IFMA instructions are prone to leak across hyperthreads, rendering the applications using FMA for optimization vulnerable.

The rest of the paper is organized as follows: Section 2 provides the necessary background information. Section 3, 4 and 5 are dedicated to the Shesha framework and its discoveries. In Section 6, we perform a root-cause analysis of the discovered transient execution paths and then discuss how they can aid as building blocks to micro-architectural attacks in Section 7. In Section 8, we show practical exploitation of leakage from FMA instructions. We then provide a discussion on related works in Section 9 and mitigation strategies in Section 10. Finally, we conclude in Section 11.

Responsible Disclosure: We have disclosed the results of this work, including the leakage due to FMA execution unit to Intel. Intel acknowledged the leakage from the FMA execution unit, and responded that mitigation from [25] could cover this issue as well; therefore, no embargo period is required.

Artifact Availability: The tool and proof-of-concept attack codes can be found at https://github.com/SEAL-IIT-KGP/shesha.

2 Background

2.1 Superscalar Computing

High-performance CPUs, such as Intel Core and Xeon processors, utilize multiple cores, high-speed memory units, and support parallel processing with efficient isolation and security measures. Memory isolation is achieved through individual per-process virtual address spaces. These virtual addresses are translated to page table entries (PTEs) containing data location, access control, and status bits for access control. These modern CPUs employ Simultaneous Multithreading (SMT) that allows multiple threads to execute on the same core, providing architectural isolation. Intel CPUs support two threads to run simultaneously per physical core. Another important feature called speculative execution enables the CPU core to execute instructions in the pipeline even when there are dependencies on prior unresolved operations. In case of incorrect predictions, the CPU corrects itself by re-executing instructions. While architecturally invisible, there may be observable side effects due to microarchitectural state changes. In this context, such an instruction is referred to as a *transient instruction* [9, 30, 33]. Transient execution attacks capitalize on these transient instructions to leak sensitive data across security boundaries. Single Instruction, Multiple Data (SIMD) enables data-level parallelism, performing the same operation on multiple sets of data. AVX2 and AVX-512 are key SIMD extensions for x86 architecture. Finally, the CPU includes a shared last-level cache (LLC) across execution cores and an interconnect bus linking the LLC, cores, and DRAM. Additionally, each core has its own private caches - L1 and L2 caches. Temporal buffers are employed to optimize micro-operations, such as the fill buffer for fetching cache line data bits and the store buffer for holding data before committing it to the cache.

2.2 Particle Swarm Optimization

A Particle Swarm Optimizer [28], abbreviated PSO, is a generic evolutionary method suited to solving optimization problems. Unlike Genetic Algorithms that rely upon evolution of genetic information, PSOs work on the delicate interplay of individual and swarm behaviours in nature (like ant swarm, bee swarm etc). Abstractly, a PSO balances two opposing forces- *exploration* and *exploitation* in order to optimize an objective function. Here, *exploration* abstracts the random choices that encourage the PSO for a random walk across the search space in hopes of finding newer paths to global optimums. On the other hand, *exploitation* abstracts the greedy choices, encouraging the PSO to focus on a direct path to currently known optimums. As evident, too much of *exploration* is no better than a random search, while too much of *exploitation* risks getting stuck in local optimums. A PSO balances the two forces by switching back-and-forth between *particle* and *swarm* behaviour. A particle performs random searches across the search space, and if optimal paths are discovered, broadcasts that information to other particles, such that the entire swarm (where swarm can be interpreted as the set of all particles) starts to use the information from that one particle. Consequently, the particle behaviour abstracts exploration, while the swarm behaviour abstracts exploitation.

Syntactically, a PSO is characterized by a *fitness* function $f : \mathcal{R}^n \rightarrow \mathcal{R}$, mapping an n-dimensional vector (termed *particle position*, or $\mathbf{x}$) to a real-value (termed *fitness*) which expresses how close the particle position is wrt. global optimum. Initially, the PSO samples a random *population* of N particles, denoted by a set of position vectors $\mathbb{X} = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, ..., \mathbf{x}_N\}$. For an iteration i of the PSO algorithm and particle $j : \{1 \leq j \leq N\}$, the terms *velocity* and *position* are defined as:

$$\mathbf{v}_j(i+1) = \alpha.\mathbf{v}_j(i) + \beta.(\mathbf{p}_j(i) - \mathbf{x}_j(i)) + \gamma.(\mathbf{p}_g(i) - \mathbf{x}_j(i))$$
$$\mathbf{x}_j(i+1) = \mathbf{x}_j(i) + \mathbf{v}_j(i+1)$$

where α is the inertial weight, β is the cognitive acceleration coefficient, γ is the social acceleration coefficient, $\mathbf{p}_j$ is the best position (where the fitness function is maximally optimal, up to iteration i) of particle j, and $\mathbf{p}_g$ is the best position for the entire swarm. Abstractly, in every iteration $i+1$, each particle j attempts to *update* its position $\mathbf{x}_j(i+1)$ based on the position in the previous iteration $\mathbf{x}_j(i)$ and a computed *velocity* $\mathbf{v}_j(i+1)$. Now, this velocity vector is computed by combining three pieces of information:

1. **Velocity in the previous iteration**: denoted by $\mathbf{v}_j(i)$
2. **Exploration / Particle behaviour**: Velocity computed by considering the best position explored by the particle so far (denoted by the vector $\mathbf{p}_j(i)$).
3. **Exploitation / Swarm behaviour**: Velocity computed by considering the best position across the entire swarm (denoted by the vector $\mathbf{p}_g(i)$).

The parameters α, β, and $\gamma \in [0,1]$ control the importance given to each of these three information points, and directly affect how a particle moves across the search space.

3 Shesha: Automatic Analysis of Bad Speculation in Instruction Set Extensions

In this section, we put forward a Particle Swarm Optimizer (PSO) inspired framework to perform an optimized exploration of search space related to bad speculation in ISEs. Towards this, we first elaborate on how the search space of bad speculation is *disjointedly fragmented*, allowing us to make modifications to generic PSO semantics (cf. Section 2.2) that provide faster convergence rates. Concretely, following the semantics of Section 2.2, instead of relying upon the entire swarm for swarm behaviour, the algorithm need only rely upon fragmented portions of the swarm, thereby allowing

Table 1: Performance counters reporting on occurrences of bad speculation. (TMA: Top-down Microarchitecture Analysis)

Name	Description	Class of bad speculation
ASSISTS.FP	Counts all microcode Floating Point assists.	Microcode assist
ASSISTS.HARDWARE	Counts all hardware assists.	Microcode assist
ASSISTS.PAGE_FAULT	Counts all assists related to page faults.	Microcode assist
ASSISTS.SSE_AVX_MIX	Counts all assists related to AVX-SSE transitions.	Microcode assist
MACHINE_CLEARS.DISAMBIGUATION	Counts machine clears due to disambiguation issues	Machine clear
MACHINE_CLEARS.MEMORY_ORDERING	Counts machine clears due to memory ordering issues	Machine clear
MACHINE_CLEARS.SMC	Counts machine clears due to self-modifying code	Machine clear
BR_MISP_RETIRED.ALL_BRANCHES	Counts retired, mispredicted branch instructions.	Branch misprediction
TOPDOWN.BR_MISPREDICT_SLOTS	Counts TMA slots wasted due to branch misprediction	Branch misprediction

more granularity and enabling faster convergence. The disjointedly fragmented nature of the search space allows splitting Shesha into two phases: ① Cognitive phase (where there is no social acceleration), and ② Mixed phase (partial social/cognitive acceleration). By end of phase ①, each particle of the swarm is present in one of the disjointedly fragmented sub-space, and has knowledge of other particles in the same subspace. Thereby, in phase ②, the social acceleration component can be focused only on the fragmented sub-space. We detail this modification to the PSO semantics further in this section, along with the threat model.

3.1 Threat Model

Since the objective of Shesha is to *explore* possible avenues of transient execution attacks, we assume Ring 3 `sudo` privileges. However, for attack demonstration, we do not require `sudo` privilege and only assume user-level privilege. We further assume x86-64-based target systems with latest microcode patches and secure (patched) operating system, with all state-of-the-art mitigations against transient execution attacks. We assume no publicly known exploitable vulnerability is present in the system. We specifically focus on Intel CPUs with newer generation microarchitecture, both on client and server platforms. For our experiments, we execute Shesha on Intel CPUs from four generations: Alder Lake(12th gen Core i7) and Comet Lake (11th gen Core i7) on the client side and Sapphire Rapids (4th gen Xeon) and Ice Lake (3rd gen Xeon) on the server side. All subsequent discussions are contextual to these aforementioned machines.

3.2 Fitness function

As discussed in Section 2.2, the objective of a PSO is to optimize a well-defined fitness function $f : \mathcal{R}^n \rightarrow \mathcal{R}$. Since our search space is instruction sequences enabling bad speculation over ISEs, our fitness function must be a combination of indicators that capture the existence or absence of bad speculation. For a code sequence $\mathcal{C}$, in our context, f can be defined as $f : \mathcal{C} \rightarrow \{0,1\}$, or f provides a binary decision whether bad speculation was observed for a given code sequence $\mathcal{C}$. We defer discussion on the generation of $\mathcal{C}$ to Section 3.4.1 and focus on the binary decision of bad speculation here.

First, we detail on when bad speculation happens in modern Intel processors. Intel's Software Developer Manual [24] states bad speculation occurs from three major causes - ① branch misprediction, ② microcode assists, and ③ machine clears. Branch misprediction related transient execution occurs due to the need to squash micro-operations issued from the mispredicted branch and re-issue the ones from the correct branch. Likewise, microcode assists are needed when the hardware needs to rely upon microcode for execution. In such a scenario, the speculatively issued micro-operations from the micro-operation decoder are flushed from the pipeline, and are replaced by the micro-operations from the microcode sequencer (which in turn fetches these micro-operations from microcode ROM). Finally, machine clears also cause transient execution for events like memory ordering violations, self-modifying code execution, and so on.

Given this context, to implement the fitness function $f : \mathcal{C} \rightarrow \{0,1\}$, we require an interface to capture when any of ① branch mis-prediction, ② microcode assist, or ③ machine clear happens. All modern Intel systems expose a Performance Monitoring Unit[1] that logs different types of events occurring on the physical cores for analysis. Such performance counters are useful in understanding occurrences of bad speculation, given any code sequence $\mathcal{C}$. Table 1 summarizes the exact performance counters we used for Shesha. Therefore, given a code sequence $\mathcal{C}$, the objective of the PSO will be to *maximize* occurrences of bad speculation as reported by the respective performance counters.

3.3 Disjointedly Fragmented Sub-spaces

We now discuss the nature of bad speculation events that we term as *disjointedly fragmented* sub-spaces which allows us to achieve faster convergence for Shesha and efficiently scale as the number of ISE instruction pool increases. As evident from Table 1, each of the performance counters pertains to bad speculation scenarios, which are **disjoint** from each other from a causal perspective. For instance, bad speculation due to

[1] Intel Perfmon events. https://perfmon-events.intel.com/

memory disambiguation issues has completely different cause (incorrect speculation while resolving load/store addresses) than bad speculation due to self-modifying code execution (incorrect speculation in writing to instruction cache). Additionally, the disjoint classes of bad speculation also have effects on disjoint architectural and micro-architectural assets. For example, memory disambiguation issues will mainly cause speculation in data forwarding in load/store buffers. On the other hand, self-modifying code execution issues will affect the instruction cache. Additionally, AVX-SSE transition issues shall affect the SIMD registers and temporal buffers. Therefore, we make the following observation:

Equivalence Classes: *Different scenarios of bad speculation in Table 1 are* ***disjoint*** *wrt. cause and effect (on architectural/micro-architectural assets), thereby splitting the search space of bad speculation into disjointedly, fragmented* ***equivalence classes****, where the cause/effect of one equivalence class are unaffected by that of other classes.*

Such a definition of *equivalence classes* (over set of instructions) ensures that the search space for bad speculation is split into disjointedly fragmented sub-spaces, such that the portion of the PSO's swarm working on one equivalence class need not rely upon the entirety of the PSO swarm for exploitation (cf. Section 2.2). For example, without loss of generality, consider a portion of the PSO swarm working with machine clears related to self-modifying code. Then, the velocity vector computations related to swarm behaviour (i.e. involving $\mathbf{p}_g(i)$, cf. Section 2.2) need not consider the portion of the swarm working with bad speculation due to branch mispredictions, simply because code sequences triggering branch misprediction (for instance, heavy use of loops to train the branch predictor) have no direct bearing with code sequences triggering self-modifying code (for instance, use of store operations in address ranges belonging to `.text` section of the process binary, thereby requiring write operations on the instruction cache). Based on this description of equivalence classes, we now detail the construction of Shesha in the next subsection.

3.4 Swarm Optimizer description

With the idea of equivalence classes established, we now detail Shesha - an automatic analysis framework for exploring the disjointedly fragmented equivalence classes of bad speculation in ISEs. Shesha is divided mainly into three phases: ❶ Swarm initialization, ❷ Cognitive phase, and ❸ Mixed phase. In phase ❶, a population pool of `N` particles is created where each particle represents a randomly sampled instruction sequence from the instruction pools of different ISEs. Then, in phase ❷, the particles are evolved to settle into sub-swarms (i.e. one swarm for one equivalence class). In this phase, since the sub-swarms are still being created, there is no exploitative behaviour (i.e. all particles evolve independently of each other). This design decision is intentional, following our discussion on equivalence classes in Section 3.3. Until the sub-swarms are ready, Shesha does not have reliable global best solutions to use for updating particle velocity (cf. Section 2.2 for PSO semantics). Finally, upon end of phase ❷, the sub-swarms would be ready for each of the equivalence classes, which is when phase ❸ kicks in and uses both *local* best and *global* best solutions to update particle velocity. We now explain these phases in detail.

3.4.1 Swarm Initialization

The first phase initializes a population of N particles, where each represents an instruction sequence, sampled from the ISE instruction set. Following the semantics from Section 2.2, each particle is represented by a position vector $\mathbf{x}$ in a n-dimensional real-valued vector space, such that the fitness function $f : \mathcal{R}^n \rightarrow \mathcal{R}$ can be evaluated upon the same. We describe now how we map a code sequence to a real-valued n-dimensional vector.

To achieve this, we leverage a machine-readable `x86` ISE list from uops.info [1] and use it to construct the ISE instruction set pool, further sub-divided into about 90 ISE types, like `AES`, `AVX`, `AVX2`, `AVX512EVEX`, `AVX512VEX`, `AVXAES`, `BMI1`, `BMI2`, `SSE`, `SSE2`, `SSE3`, `SSE4`, `X87` to name a few. Shesha allows the user to choose which ISEs to be explored. To generate a single particle, Shesha randomly samples n instructions from the selected instruction pool as well as randomly selects operands (for instance, which registers to use out of the entire register bank). Following the description of fitness function discussed in 3.2, this ordered set of n sampled instructions and their operands corresponds to the instruction sequence $\mathcal{C}$. In our experiment, we set the values of N and n as 50 and 10, respectively (without loss of generality).

Next, we derive a position vector $\mathbf{x}$ in the n-dimensional real vector space from $\mathcal{C}$. To do so, we first assign a unique real-value `opcode` to each instruction in the entire set of ISEs considered by Shesha. Likewise, the operand description of the instruction `opcode` is also assigned a uniquely identifiable real-value `operand_identifier_bitstring`. The `opcode` and the `operand_identifier_bitstring` can be combined to uniquely identify any instruction amongst all possible instructions in the ISEs under consideration. We note here that `operand_identifier_bitstring` depends simply on the operands used, and *not* on the actual data held by the operands when the instruction is executed. Finally, the *combined* representation of an instruction is uniquely identifiable by `(opcode « i) | operand_identifier_bitstring`, where i is the number of bits needed to uniquely identify the operands. We explain the mapping using an example. Assume the sampled instruction is `VPXOR xmm1, xmm2, xmm3`, where `VPXOR` is the AVX flavor of Logical XOR on SIMD registers `xmm1`, `xmm2`, `xmm3`. Now `VPXOR` shall have a unique `opcode` iden-

tifying this instruction, while each of the operand registers can be uniquely identified by a 4 bit value (since there are 16 unique `xmm` registers). Hence, the real-valued unique identifier for this instruction shall be `(vpxor_opcode « 12) | (1 « 8) | (2 « 4) | 3`, assuming `xmmk` register to be uniquely identifiable by the 4 bit integer k. Repeating the same procedure for all sampled instructions creates n distinct real values, which contribute to the n-dimensional position vector $\mathbf{x}$. At the end of this phase, we have the set of position vectors $\mathbb{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$. Next, we describe how to compute velocity vectors in order to mutate these position vectors.

3.4.2 Cognitive phase

In this phase, we allow for the *compartmentalization of the swarm population* into the disjointedly fragmented equivalence classes. Since the initialization phase has already computed the set of position vectors $\mathbb{X} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\}$, we focus on generation of velocity vectors that allow generation of sub-swarms (i.e. one sub-swarm for each equivalence class). In order to do this, we must prevent any exploitation/swarm behaviour (cf. Section 2.2) from kicking in the cognitive phase. For this case, we choose $\alpha = 1$, cognitive acceleration coefficient β as 0.4, and social acceleration coefficient γ as 0 (to cancel out any swarm behaviour). This forces using the local best solutions $\mathbf{p}$ with probability β. In this phase, we rely upon two kinds of mutations to the velocity vector:

1. **Modification to operands**: Given a particle position vector $\mathbf{x}_j : 1 \leq j \leq N$, we sample a random dimension $d : 1 \leq d \leq n$ and mutate the operands with probability β. Following the semantics of 3.4.1, assume $x_j^d =$ `(opcode « i) | operand_identifier_bitstring`), where | represents Logical OR. Thereby, the new position vector is evaluated as $\mathbf{x}'_j = \{x_j^1, x_j^2, ..., x_j^{d-1}, (x_j^d \wedge$ `(1 « i)) « i | new_operand_identifier_bitstring`, $x_j^{d+1}, ..., x_j^n\}$. Here, the newer operands that are sampled are converted to `new_operand_identifier_bitstring` using the same semantics as in Section 3.4.1. To follow upon the same example of `VPXOR xmm1, xmm2, xmm3` having representation `(vpxor_opcode « 12) | (1 « 8) | (2 « 4) | 3`) in the position vector, assume the operand mutation converted this instruction to `VPXOR xmm5, xmm13, xmm15`. Then the modified representation of the aforementioned instruction in the position vector can be given by `(vpxor_opcode « 12) | (5 « 8) | (13 « 4) | 15`).

2. **Modification to instruction**: Like operand modifications, here also $x_d =$ `(opcode « i) | operand_identifier_bitstring`) is mutated as $x'_d =$ `(new_opcode « i) | operand_identifier_bitstring`) with probability β (cognitive acceleration coefficient), where `new_opcode` is a newly sampled instruction from the ISE set. Again taking the same example, suppose `VPXOR xmm1, xmm2, xmm3` is mutated to `VPAND xmm1, xmm2, xmm3`, then `(vpxor_opcode « 12) | (1 « 8) | (2 « 4) | 3`) shall be mutated to `(vpand_opcode « 12) | (1 « 8) | (2 « 4) | 3`).

Finally, the mutated position vector is evaluated upon the fitness function as defined in Section 3.2 to understand if the mutated particle discovers some newer avenues of bad speculation. As iterations of the Cognitive phase increase, the entire population of particles converges upon one of the several disjointedly fragmented equivalence classes. Thereby, at the end of the Cognitive phase, the sub-swarms belonging to each equivalence class $\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_9$ are output to phase ❷ (where each S_i belongs to one equivalence class as in Table 1). Concretely, each particle in the initialized swarm belongs to exactly one of these equivalence classes.

3.4.3 Mixed phase

In this phase, we enable swarm behaviour onto the discrete sub-swarms $\mathcal{S}_1, \mathcal{S}_2, ..., \mathcal{S}_9$ output from the Cognitive phase. Concretely, the Mixed phase makes the following changes to Cognitive phase settings: ① reduced rate of exploration, ② increased rate of exploitation from the sub-swarm $\mathcal{S}_i$ to which the particle belongs to. To achieve this, we set $\alpha = 1$, cognitive acceleration coefficient $\beta = 0.1$ (as opposed to 0.4 in the Cognitive phase), and the social acceleration coefficient $\gamma = 0.4$ (as opposed to 0 in the Cognitive phase). This setting still allows exploration using the local best solutions, albeit with reduced probability β, while also starting to use the global best solutions found in the sub-swarm $\mathcal{S}_i$ to which the particle belongs. In this phase, we rely upon three kinds of mutations to the velocity vector:

1. **Modification to operands**: Same as Cognitive phase, but with reduced probability $\beta = 0.1$ instead of 0.4.

2. **Modifications to instruction**: For each particle position vector $\mathbf{x}_j$ belonging to some sub-swarm $\mathcal{S}_k$, sample two random dimensions $d_1, d_2 : 1 \leq d_1, d_2 \leq n$ and mutate $x_j^{d_1}$ as in Cognitive phase with probability $\beta = 0.1$. For dimension d_2, with probability $\gamma = 0.4$, replace the d_2-th dimension of $\mathbf{x}_j$ with the d_2-th dimension of the leader of sub-swarm S_k. Here, *leader* of the sub-swarm $\mathcal{S}_k$ is defined as the position vector in sub-swarm S_k with maximum fitness as described in Section 3.2 (i.e. the instruction sequence with maximum bad speculation in the equivalence class $\mathcal{S}_k$).

3. **Dimensionality Reduction**: For a given position vector $\mathbf{x}_j : 1 \leq j \leq N$ belonging to sub-swarm S_k and defined as $\mathbf{x}_j = \{x_j^1, x_j^2, x_j^3, ..., x_j^n\}$, remove a randomly sampled dimension $d : 1 \leq d \leq n$ with probability $\beta = 0.1$. One might recall that each element in $\mathbf{x}_j$ is representative of

Table 2: Leakage Variants Discovered by Shesha

Cases	Operations	Register Dependency	Equivalence Class	Assets Affected	Generation	Window Size	Attack Type
①	SIMD-Vector intermixing	×	SMC, SAM	I-Cache, SIMD registers	12G ✓, 11G × 4X ✓, 3X ×	17 (1 `rep`)	Leak
②	Single and Double precision intermixing	✓	HW, SMC, MO	I-Cache, precision conversion hardware, SIMD buffer	12G ✓, 11G × 4X ✓, 3X ×	20 (100 `rep`)	Leak
③	Fused Multiply and Addition	×	FPA	FPU	12G ✓, 11G ✓ 4X ✓, 3X ✓	12 (32 `rep`)	Leak, Injection
④	SSE-AES using Denormal numbers	✓	FPA, SMC	FPU, AES hardware	12G ✓, 11G ✓ 4X ✓, 3X ✓	12 (32 `rep`)	Leak, Injection

SMC: Self-Modifying Code; SAM: SSE-AVX Mix; HW: Hardware Assist; FPA: Floating Point Assist; MO: Memory Ordering; rep: No. of repetitions
Leak: Transient Leakage through Flush+Reload; Injection: Floating Point Value Injection; ✓: Required or Available; ×: Not required or Not available
FPU: Floating point unit; 12G: Alder Lake; 11G: Comet Lake; 4X: Sapphire Rapids; 3X: Ice Lake

an instruction. Hence, informally, we can treat the real-valued vector $\mathbf{x}_j$ as an instruction sequence $\mathcal{C}$ comprising of n instructions. Therefore, dimensionality reduction aims to *reduce* this instruction sequence so as to generate smaller and focused sequences, while still maximizing the occurrence of bad speculation.

At the end of this phase, Shesha outputs the population position vectors with maximized fitness (i.e. maximum occurrence of bad speculation) in the respective equivalence classes. We note that dimensionality reduction, in particular, ensures that the final output of Shesha is able to drop instructions from the generated instruction sequence that may not contribute to bad speculation. In other words, because of dimensionality reduction, the final output of Shesha is a *minimal* reproduced instruction sequence responsible for the relevant bad speculation scenarios detailed in Table 1.

4 Uncovered Transient Execution Paths

We ran a test campaign of 24 hours, in which most observations were obtained within the first 8 hours (refer to Algo. A in Appendix for description of the algorithm). Since we rely upon Intel Performance Monitoring Unit (PMU), which captures the *events* occurring in the system with certainty, the false positive rate of Shesha is zero. In this section, we detail the novel transient execution paths uncovered by Shesha as a result of the testing campaign. We also reverse engineer the root-cause of these transient execution paths and detail their effect on the overall behaviour of the system under test.

1. **Hardware assists for precision conversion**: All floating point numbers can have two precision levels: single or double. With dedicated instructions for single precision and double precision separately, the same set of registers (i.e. {`xmm0, xmm1, ...`}, {`ymm0, ymm1, ...`} and {`zmm0, zmm1, ...`}) need to be interpreted as single precision or double precision based on the instruction operand. In this context, we uncover a **hardware assist** that kicks in case of data dependencies between instructions that intermix {`xmm0, xmm1, ...`}, {`ymm0, ymm1, ...`} and {`zmm0, zmm1, ...`} in different precision modes, where the data in the SIMD register bank needs to be interconverted between single and double precision modes.

2. **Self-modifying code execution due to intermixing precision**: Likewise, intermixing different representations of {`xmm0, xmm1, ...`}, {`ymm0, ymm1, ...`} and {`zmm0, zmm1, ...`} wrt. precision also causes **machine clears** involving updates to the instruction cache, enabling yet another transient execution path.

3. **Memory ordering issues due to intermixing precision**: We also uncover that intermixing instructions with single/double precision causes **memory ordering based machine clear**, uncovering a new transient execution path not discussed prior in literature.

4. **Intermixing SIMD-Vector instructions**. A wide variety of ISEs like `AVX`, `AVX2`, `AVX512EVEX`, `AVX512VEX`, `SSE`, `SSE2`, `SSE3`, `SSE4`, `SSE4a`, and `SSSE3` share the SIMD register banks {`xmm0, xmm1, ...`}, {`ymm0, ymm1, ...`} and {`zmm0, zmm1, ...`}. As such, transitioning from SIMD to Vector instructions or vice-versa incurs a machine clear attributed to self-modifying code. We note that although older Intel generation processors like Skylake had transition penalties related to intermixing of SIMD-Vector instructions [24], recent Intel processors up to Comet Lake have no such penalties [40]. However, as we uncover, the 12-th generation client processor (Alder Lake) and 4-th generation Xeon processor (Sapphire Rapids) show transient execution due to an involved **machine clear due to self-modifying code**.

5. **FP assist due to Denormal arithmetic**: We also uncover newer avenues of **Floating Point assists** in the Fused-Multiply-Add instruction family as well during

transitions between AES-NI and SSE instructions. Although Floating Point assists have been demonstrated before [40], our tool and analysis show that Floating Point assists have a wider scope than simple SIMD arithmetic (as previously discussed in literature).

5 Shesha Design Choices

We now explain the choice of parameters by comparing the role of Particle behaviour and Swarm behaviour in Shesha. To do so, we define 6 variants of Shesha wrt. hyperparameters β (which controls Particle behaviour) and γ (which controls Swarm behaviour). To compare between the variants, we define two metrics: ① time taken to find the *first* SIMD-Vector speculation, and ② time taken to find the *first* precision intermixing speculation. The reason for this (as detailed in subsequent sections) is because SIMD-Vector speculation is the easiest of the four (cf. Table 2) to discover. This is because of the large number of instructions under these ISEs, as well as no special requirement (like read-after-write dependencies, or specific floating-point arithmetic) needed to trigger this speculation. Precision intermixing, however, is on the other extreme, and is the hardest of the four to uncover. This is because (as we detail in Section 6.2) such speculation requires closely tied instructions that handle single/double precision respectively, and have read-after-write dependency.

Variants Shesha$_{\{\beta=1,\gamma=0\}}$ and Shesha$_{\{\beta=0,\gamma=1\}}$ are essentially equivalent to random sampling, with different sources. The former prioritizes mutating the chosen instruction within the particle itself by replacing it with a random instruction with probability 1. The latter also works to similar goal, but replaces the chosen instruction with another instruction from other particle in the swarm. Then, variants Shesha$_{\{\beta=0.4,\gamma=0\}}$ and Shesha$_{\{\beta=0.1,\gamma=0\}}$ test the extent of particle behaviour by changing the probability β of replacing a chosen instruction with a randomly sampled instruction. Finally, variants Shesha$_{\{\beta=0.1,\gamma=0.1\}}$ and Shesha$_{\{\beta=0.1,\gamma=0.4\}}$ test the extent of swarm behaviour by changing the probability γ of replacing a chosen instruction with an instruction from another particle in the swarm. We summarize these in Table 3.

First, we note that Shesha$_{\{\beta=0.1,\gamma=0\}}$ is inherently slow in both aspects we consider here: it adopts a *random* sampling strategy (by setting $\gamma = 0$), and performs such sampling at a reduced rate of 0.1. The best performance (wrt. Precision Intermixing) is given by Shesha$_{\{\beta=0.1,\gamma=0.4\}}$, which allots a slightly higher probability of swarm behaviour (through $\gamma = 0.4$) than other variants. However, it is the variant Shesha$_{\{\beta=0.4,\gamma=0\}}$ that outperforms all wrt. SIMD-Vector intermixing, possibly due to balancing the aggressiveness with which to replace instructions. Interestingly, note that variants equivalent to random sampling (i.e. Shesha$_{\{\beta=1,\gamma=0\}}$ and Shesha$_{\{\beta=0,\gamma=1\}}$) perform fairly poorly wrt. SIMD-Vector intermixing (possibly due to very aggressive mutation strategy).

As a result, in our final deployment of Shesha, we adopt a mixture of variant Shesha$_{\{\beta=0.4,\gamma=0\}}$ and variant Shesha$_{\{\beta=0.1,\gamma=0.4\}}$. The first variant forms the Cognitive phase (cf. Section 3.4.2), while the second variant forms the Mixed phase (cf. Section 3.4.3). Finally, we note that variant Shesha$_{\{\beta=1,\gamma=0\}}$ *simulates* other *fuzzing* (essentially relying upon a random sampling strategy) based tools like [36, 38, 39, 55] had they supported ISEs like AVX/SSE/FMA and so on. For comparison however, we report the raw data from these works, albeit with different objectives over mostly x86 ISA. For instance, wrt. the speculative paths and ISA explored by [38], it is able to detect violations in about 5 minutes (which is comparable to Shesha$_{\{\beta=0.4,\gamma=0\}}$'s detection time for SIMD-Vector transient path detection). Likewise, in [36], the entire fuzzing campaign lasts about 20+ CPU hours. On similar lines, the detection in [39] ranges from a minute to about an hour based on the type of speculation targeted [2]. One must note that the objectives of each of these tools (as presented in their corresponding papers) were different, and thus, they explore different subsets of ISAs. Shesha works with comparatively larger ISA set and thus explores wider range of instruction combinations and stimulates variety of specialized execution units and buffers.

Table 3: Time taken to uncover speculation by the variant.

Variant	SIMD-Vector	Precision Intermixing
Shesha$_{\{\beta=1,\gamma=0\}}$	0.5 hour	14 hours
Shesha$_{\{\beta=0.4,\gamma=0\}}$	0.01 hour	8 hours
Shesha$_{\{\beta=0.1,\gamma=0\}}$	1 hour	$\approx$ 24 hours
Shesha$_{\{\beta=0.1,\gamma=0.1\}}$	0.6 hour	4 hours
Shesha$_{\{\beta=0.1,\gamma=0.4\}}$	0.03 hour	2 hours
Shesha$_{\{\beta=0,\gamma=1\}}$	0.5 hour	9 hours

6 Root-Cause Analysis of Discovered Transient Execution Paths

We now provide root-cause analysis for the two transition-based leakages discovered by Shesha, namely, SIMD-Vector transition and single-double precision transitions.

6.1 SIMD-Vector Transition based SMC

SIMD and Vector instruction extensions like `AVX`, `AVX2`, `AVX512EVEX`, `AVX512VEX`, `SSE`, `SSE2`, `SSE3`, `SSE4`, `SSE4a`, and `SSSE3` are central to several performance gains experienced in modern Intel processors. These ISEs allow greater than 64 bit registers holding more than one data point, to be operated in parallel by the same instruction (hence the name SIMD type ISEs). To do so, modern IA-64 architectures define a register bank with sixteen 512-bit wide registers. Interestingly, ISEs like `AVX2`/`SSE` allow *differing* representations of the *same* hardware register in two dimensions: ① width w of the register, and ② the SIMD repre-

[2] [55] does not report convergence time.

sentation of the data contained in that width *w*. To elaborate on dimension ①, the *same* hardware registers can be interpreted as 512-bit wide registers `zmm0, zmm1, zmm2, ..., zmm15` (by ISEs like `AVX512EVEX`, `AVX512VEX`), as 256-bit wide registers `ymm0, ymm1, ymm2, ..., ymm15` (by ISEs like `AVX2`), and as 128-bit wide regisers `xmm0, xmm1, xmm2, ..., xmm15`. Likewise, regarding dimension ②, the same data in any `zmm`/`ymm`/`xmm` register can be interpreted as differently packed integer/floating-point value. This discussion raises a concerning question: *Given a processor allowing executions of ISEs requiring different representations of the same hardware, how are transitions between such ISEs handled*? According to the Intel Software Developer Manual [24], the exact implementation to handle such transitions depends upon the exact micro-architecture under question. We make the following observation as the root cause of bad speculation (as in Table 2) encountered in modern Intel processors when transitioning from SIMD to Vector instructions (and vice-versa)[3].

Observation. *By deciding to remove the Finite State Machine based register dependency analysis and insertion of blend operations into the issue state of the pipeline, micro-optimizations for Alder Lake and Sapphire Rapids handle SIMD-Vector transitions by incurring a self-modifying code execution that inserts the blend operations directly into the instruction cache, therefore causing speculation.*

6.2 Precision based Transient Execution Paths

Modern ISEs usually work upon a register bank of 16 registers of width 512 bits, and the same data on these registers can be interpreted in different ways. While it is straightforward to switch between integer representations, switching between single-precision and double-precision representations is an involved process. Through the course of our testing campaign, Shesha uncovered a previously unknown source of transient execution paths: *non-explicit switching between single and double precision floating-point numbers in a Read-after-Write (RAW) data dependency*. Concretely, a sequence of two instructions linked by a RAW dependency causes novel transient execution paths on Intel CPUs if they require contrasting representations (single precision vs double precision) of the same floating-point number. Interestingly, three independent, completely novel transient execution paths are uncovered by Shesha as a result of single/double precision intermixing with RAW data dependency- ① hardware assist (HW), ② self-modifying code (SMC) based machine clear, and ③ memory ordering (MO) violation based machine clear. For all subsequent discussions, we consider the following abstract instruction sequence (`SP`: single precision instruction, `DP`: double precision instruction):

1. `SP xmm0, xmm1, xmm2` followed by `DP xmm3, xmm4, xmm0` **OR**
2. `DP xmm0, xmm1, xmm2` followed by `SP xmm3, xmm4, xmm0`

Precisely, for the discussed transient paths to kick in, the requirement is a set of at least two instructions with a Read-after-Write (RAW) dependency connecting them (using the `xmm0` register in the above examples), without occurrence of an explicit conversion instruction (like `CVTPD2PS` or `CVTPS2PD`) in between. We note here that Write-after-Write (WAW) or Write-after-Read (WAR) dependencies do *not* trigger any of the following transient paths, simply because there is no need for any implicit precision conversion as the second instruction is *writing* to the dependent register and not reading from it. Writes from a single-precision instruction to a register holding a double-precision value are architecturally simply *overwriting* the registers, and thus incur no transient executions. We summarize the following observation as the root cause of all the bad speculation events when transitioning between single and double precision[4].

Observation. *Owing to differing representations of same data in `xmm`/`ymm`/`zmm` wrt. single and double precision, a read-after-write (RAW) dependency without `CVTPD2PS`/`CVTPS2PD` incurs a hardware assist to perform implicit conversion, hence triggering aggressive speculation. Moreover, accompanying machine clears related to self-modifying code and memory ordering are also involved, thus contributing to speculation.*

7 Attack Building Blocks

7.1 Transient Execution Window

Transient execution paths guarantee flushing of the in-flight micro-operations from the pipeline and re-steer the execution on the correct path. As such, the *width* of the transient window for any given transient execution path determines the number of micro-operations an attacker can issue on the transient path before the rollback of the instructions. Practically, a larger transient window allows for executing complex attack vectors in the transient path. Architecturally, no effect of any transient micro-operation is visible. Therefore, we need to rely upon micro-architectural effects to measure the transient window. In our case, we compute the transient window by counting the number of transient loads executed before the pipeline rollback. When no bad speculation (discovered by Shesha) occurs, we observe no activity in the transient path, implying absence of misprediction-based transient execution. Our results are summarized in Table 2. In comparison to the machine

[3] For details of these implementations and our reverse-engineering approach, please refer to Appendix B of the full version of the paper [10].

[4] For a detailed discussion on each of these transient execution paths and our reverse-engineering approach, please refer to Appendix C of the full version of the paper [10].

clears presented in [40], we conclude that the SIMD-Vector transition-based SMC exhibits a larger transient window than all other variants of transient executions through machine clears or assists (both uncovered here as well as in [40]). This observation directly ties in with the number of *repetitions* (for warming up the execution pipeline) an attacker needs to do on the transient path before the actual attack can be mounted (cf. Section 7.1). As evident, due to the largest transient window with minimum number of repetitions, SIMD-Vector transition-based SMC provides a very practical transient execution path (with effectively no repetitions required), thereby becoming an actual choice for an adversary to mount in-the-wild attacks.

Enlarging the transient window. As different transient execution paths have differing transient windows, not all of them would be directly suited to leak reliably. To circumvent this, a usual method is to warm up the pipeline before the actual attack; this can be done solely on the attacker's end. This can be done by several repetitions (`rep`) of the same transient operations to maximize the transient window. To determine the extent of pipeline warm-up needed to leak data, we increase the repetitions of warm-up until we observe leakage from the Flush+Reload [57] covert channel. In Table 4, we provide a comparative analysis of the number of repetitions needed to warm up the pipeline. Lesser the number of required repetitions `rep`, more likely it is to find in-the-wild occurrence of the uncovered transient paths in in-the-wild code bases [5].

Table 4: Comparative analysis of the extent of pipeline warmup needed by different transient paths. Leakage rate is computed as the ratio of number of transient loads to the `rep` needed to observe visible differences in Flush+Reload [57].

Transient Path	Required `rep`	Leakage Rate
FP assist [40]	32	0.375
SMC assist [40]	20	0.3125
Snoop MO [40]	150	0.4
MD [40]	32	0.53125
Page Fault [35, 36]	1	0.1
SIMD-Vector (**Our work**)	1	17
D2S/S2D (**Our work**)	100	0.2
FMA/AES (**Our work**)	28	0.42

7.2 SMT Covert Channel

We demonstrate a covert channel using contention in the affected hardware assets. Two processes are pinned to two logical cores of the same physical core. Hence, we assume Simultaneous Multi-Threading (SMT) or HyperThreading to be enabled for the covert channel construction to work since the affected assets identified (in Table 2) are shared in an SMT setting. The construction of the covert channel relies upon two actors: a *sender* and a *receiver*, executing on the same physical core. The receiver *times* the execution of a code sequence $\mathcal{C}$ ($\mathcal{C}$ has no possibility of any transient execution). In contrast, depending on whether the sender has to send bit 1 or 0, it performs an additional execution involving transient execution. The presence or absence of transient execution causes contention in the concerned hardware assets, leading to a perceivable timing difference on the receiver side. Such a covert channel instantiated with SIMD-Vector transition transient executions has a bandwidth of 2 KB/s with an accuracy of 87%.

7.3 Load Value Injection

In addition to the **Leak** attack type summarized in Table 2 and detailed in Section 7.1, the Floating Point operations relying upon an assist to carry out the denormalization operation forward incorrect *data* to subsequent transient instructions, before denormalization finishes. This scenario is a floating point flavor of classic load value injection (LVI) in transient executions [40]. However, unlike Section 7.1, the exact transient value injected into the transient operations is not architecturally visible. Hence, the adversary needs to mount LVI in two phases: ① offline phase where the adversary fuzzes the transient operation using different inputs and constructs the transient output, and ② online phase where the adversary uses the specially crafted inputs from ① that give a transient output of adversarial interest. On comparing the architectural result of the computation against the constructed LVI output, we observed both to be different, implying a successful transient value injection attack.

7.4 FP Assists due to FMA instructions

According to IEEE-754 standard, any floating point representation of a number consists of three parts: ① sign bit, ② biased exponent e, and ③ mantissa m. The standard also defines *denormalization*: a choice to trade precision for wider range of floating point numbers that can be represented. Concretely, if $e = 0$ and $m \neq 0$, denormalization can allow for a gradual underflow by appending enough leading zeroes to the mantissa until the minimal exponent is achieved.

As stated in [40], denormalization requires a Floating Point assist (denoted by `ASSISTS.FP` in Table 1), which kicks in micro-operations to *gradually underflow* a floating point number close to 0. This causes a flush of the current in-flight micro-operations from the execution pipeline, thereby creating a transient window where non-*denormalized* floating-point values are forwarded to subsequent instructions speculatively, resulting in a transient window of executions on incorrect data, while the floating-point assist is *denormalizing* the concerned value. In [40], this transient window is used to construct a flavor of load-value injection exploits on the `SSE2` instruction extension. Through Shesha however, we uncover existence of this transient execution path in other

[5] We could not test the pipeline warmup and leakage rates for [55] and [44] because of absence of Intel TSX on our test platforms.

ISEs as well [6], most notably in Fused-Multiply-Add (FMA) instructions like `VFMADD132SD`, `VFMADD132PD`, or exponentiation instructions like `VSCALEFSS`.

8 Exploiting Leakage from FMA

An interesting finding by Shesha is the transient nature of the Fused Multiply-Add (FMA) instruction set extension, which warrants a deeper investigation. Although the transient nature of AVX instructions is well-known [35, 40], the security implications of FMA (and its variants) instructions have not been studied in literature before. In this section, we develop on the findings of Section 7.4, and focus on the exploitability of previously unreported speculation in FMA execution.

8.1 Fused Multiply-Add Execution Unit

FMA instructions have been traditionally introduced to accelerate arithmetic capabilities of Intel processors. Such instruction extensions are capable of performing *fused* arithmetic operations in one instruction execution. The classic FMA family of instructions can support operations like: ① fused multiply-add, ② fused multiply-subtract, ③ fused multiply add/subtract interleave, ④ signed-reversed multiply on fused multiply-add and multiply-subtract. Moreover, several new "FMA-like" instruction extensions have been introduced to similar effect. For instance, the Integer-FMA extension introduces two instructions that perform FMA-like operations, but on packed integers. Likewise, the 4FMAPS extension allows packing up-to four vectors in one operand (as opposed to a single vector allowed in FMA/IFMA).

Intel processors provide a specialized execution unit for FMA (and its associated sibling) extensions as detailed by the patents [5, 15]. The architecture of the hardware for executing FMA instructions poses an interesting insight: *specialized "Memory Access Units" (MAU) reside within the execution cluster of execution engine units* dedicated to FMA instruction execution. Concretely, once the *frontend* unit performs the instruction fetch and instruction decode, it issues the relevant operations to the *backend*, which consists of several units like the register file, the register renaming unit, the scheduler unit, the retirement unit, the execution unit, *and* the memory access unit. We emphasize that both the instruction cache unit as well as the instruction translation look-aside buffer (TLB) unit are a part of the *frontend*, and not the *backend*. Moreover, the MAU interfaces with both the data cache unit as well as the data TLB unit (which sit *outside* the execution engine unit, as a separate memory unit) [7].

[6] Shesha also uncovers previously unreported combinations of AES and SSE instruction sequences (like `AESDECLAST` and `MULSS` sharing a Read-after-Write register dependency) to also incur Floating-Point assists.

[7] We omit a detailed architecture description of the FMA execution unit for want of conciseness. Interested readers can refer to Intel patents [5] and [15] for more details. It must also be mentioned that these architectural descriptions depict the overall idea, and the actual implementation in the processors might differ.

8.2 Leaking arbitrary data from FMA instruction execution

As highlighted in the previous subsection, the FMA execution unit houses MAU whereas the data-cache and dTLB are kept outside the execution unit, in a stand-alone memory unit. We hypothesize that MAUs are essentially meant for *buffering optimizations*. Recall that almost all FMA (and its associated sibling extensions) have instruction variants that can take memory operations. For example, consider `VFMADD132PD zmm1, zmm2, zmm3`, which multiples the first and third operands, and then adds the product to the second operand. The other flavor of this instruction is `VFMADD132PD zmm1, zmm2, zmmword PTR [rcx]` which fetches the third operand directly from memory. Since all FMA instructions (and its sibling extensions) have flavors supporting memory operands, we hypothesize the aforementioned "Memory Access Units" or MAUs are essentially *buffers* that optimize these memory reads.

From Section 7.4, we recall that Shesha has undiscovered the possibility of speculation in FMA execution. Such speculation, tied with the existence of buffers *inside* FMA execution engine, could leak to leakages. In this context, we test the following two hypotheses related to the transient behavior shown by the FMA instructions:

❶ With speculative execution in FMA execution engine (cf. Section 7.4), does FMA "Memory Access Units" transiently forward data to speculatively executing FMA instructions?

❷ With speculative execution in SSE-AVX execution engine, does FMA "Memory Access Units" transiently forward data to speculatively executing SSE-AVX instructions?

It must be mentioned that hypothesis ❷ draws from our yet another observation about the architecture of FMA execution unit [5, 15]. The register file is shared between SSE-AVX and FMA execution engines. Concretely, the same set of `xmm/ymm/zmm` registers used by SSE-AVX instructions are also used by FMA instructions. This leads us to investigate whether any *cross-interaction* exists between these two execution engines.

All experiments are performed on 11^{th} Gen Intel(R) Core(TM) i5-11500, microcode version `0x57`, with support for FMA and IFMA extensions. We do not assume any privileges for the adversary, except for the userspace privilege to execute code.

8.2.1 Intra-FMA Execution Engine Leakage

To test hypothesis ❶, we execute victim and attacker threads on co-located CPUs. The victim thread executes FMA instructions in a tight loop. We program the victim's memory access

to some known byte (specifically `0xfa`), which is accessed by the FMA instruction. This is done to groom the MAUs within FMA execution engine with known data [8].

The attacker process forces speculation in FMA instruction (cf. Section 7.4) with memory operand. One example is `VFMADD213PD zmm1, zmm2, zmmword PTR [rcx]`), which multiplies the first and second operands and adds the value loaded by the third operand. Similar to the experiments detailed in Section 7.4, we force a floating-point assist by grooming `zmm1` and `zmm2` with carefully crafted denormal operands. Finally, we encode the entire width of `zmm1` (which is the destination register) into a Flush+Reload covert channel to leak transiently forwarded data.

In our experiments, we use the primitives detailed in Table 4 to construct and enlarge the transient window to cover speculation in FMA. Overall, we do not observe the programmed victim memory leaking through the adversarial thread's Flush+Reload channel. We attribute this absence of speculative forwarding of stale data to ① lack of speculative forwarding of stale data from MAUs to transiently executing FMA instructions, or ② the inability of constructed transient windows to cover transient execution of FMA instructions (which are inherently complex to execute). There is no evidence to favor one reason over another, though, and we leave this to future work.

8.2.2 FMA-AVX Execution Engine Leakage

To test hypothesis ❷, we follow a similar setup as Section 8.2.1 with the exception that the adversarial thread engages the AVX execution unit. Interestingly, this time, we did observe statistically significant leakage. Listing 1 shows the Flush+Reload channel activity for the victim instruction `VFMADD213PD zmm1, zmm2, zmmword PTR [rcx]`. Note how the programmed victim memory (i.e. `0xfa`) is clearly leaking to the adversary.

```
1 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa  7b  c8  c6  c8  fa fa fa fa
2 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
3 fa fa fa fa fa fa fa fa fa fa fa fa  22  8c  35  9e  ef  76  14  fa fa fa fa fa
4 d4  da  bc  e4  75  fa fa fa fa fa fa fa  cc  47  5b  11  fa fa fa fa
5 f7  b0  bf  fa fa fa fa fa fa  60  95  ad  fc  fa fa fa fa fa fa fa fa fa fa
6 fa fa fa fa fa fa fa fa fa fa fa fa fa  61  6f  40  44  b0  e8  b2  fa fa fa
7 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
8 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
```

Listing 1: Leakage when victim executes `VFMADD213PD zmm1, zmm2, zmmword PTR [rcx]`.

To test the extent of such leakage, we divided the ISEs related to FMA in the following families and tested leakage for all instructions in each family: ① `VFMADD*` (12 instructions from 3 sub-families: `VFMADD132*`, `VFMADD213*`, `VFMADD231*`), ② `VFMADDSUB*` (6 instructions from 3 sub-families: `VFMADDSUB132*`, `VFMADDSUB213*`, `VFMADDSUB231*`), ③ `VFMSUB*` (12 instructions from 3 sub-families (`VFMSUB132*`, `VFMSUB213*`, `VFMSUB231*`), ④ `VFMSUBADD*` (6 instructions from 3 sub-families (like `VFMSUBADD132*`, `VFMSUBADD213PD*`, `VFMSUBADD231PD*`), ⑤ `VFNMADD*` (12 instructions from 3 sub-families like `VFNMADD132*`, `VFNMADD213*`, `VFNMADD231*`), ⑥ `VFNMSUB*` (contains in-total 12 instructions from 3 sub-families like `VFNMSUB132*`, `VFNMSUB213*`, `VFNMSUB231*`), and ⑦ `IFMA` instructions (`VPMADD52HUQ` and `VPMADD52LUQ`).

We have confirmed the leakage in all aforementioned instructions. Moreover, other FMA-related ISEs like `4FMAPS` also follow a similar architecture as FMA and IFMA and could pose a similar risk of data leakage [9].

8.3 Attack on Cryptographic Applications

We now discuss the broader impacts of our findings. While FMA is usually used to accelerate arithmetic, it is the IFMA instruction pair that has received unprecedented adoption in recent years. This is because of IFMA instructions' capability to operate fused multiply and add operations on integers, allowing them to accelerate core operations in traditional cryptography (refer to Intel's guidelines on the same [26]). According to Intel Optimization Manual [23], IFMA is introduced for *big number multiplication for acceleration of RSA vectorized software and other Crypto algorithms (Public key) performance*. However, as a direct consequence of our findings, such optimized implementations are vulnerable to leakages, which we detail next.

8.3.1 "Almost" Montgomery Multiplication

In [17], the authors suggest using IFMA instructions to speed up modular exponentiation, which forms the basis for several critical cryptographic implementations (include RSA and ECC). This is done in two steps. First, "almost" modular multiplication (AMM) is defined algorithmically, in the sense that AMM lacks a final (conditional) reduction step (wrt. tradition modular multiplication). This allows output of one AMM call as inputs to subsequent AMM calls. Next, AMM is implemented and accelerated using `VPMADD52LUQ` and `VPMADD52HUQ` instructions. Similar to AMM, the authors in [17] also propose "almost" montgomery squaring (AMS), again accelerated using IFMA instructions. With AMM and AMS in place, modular exponentiation is effectively achieved through `VPMADD52LUQ`/`VPMADD52HUQ` instructions.

To test the extent of FMA leakage in AMM/AMS, we implement the `MulALPart` algorithm from [17] as depicted in Listing 2 (`MulAHPart` follows suit, but uses `VPMADD52HUQ` instead of `VPMADD52LUQ`). Similar to Section 8.2.2, the inputs to AMM/AMS are programmed to `0xfa`. `MulALPart` performs efficient left-shift in AMM/AMS implementation [17], such

[8] It must be noted that in an actual attack, the data would be controlled by the victim. We groom the victim's buffer in order to detect the leakage.

[9] Due to lack of support for `4FMAPS` in the CPUs we tested, we could not confirm the existence of leakage in `4FMAPS` extension family.

that this result can directly be used in other IFMA friendly operations in AMM/AMS. Note that AMM/AMS can be performed on any arbitrary integer A exceeding the width of FMA registers, thereby requiring a total of z logical registers (notated as $\{A_1, A_2, ..., A_z\}$). Likewise, i represents the index of the *current* chunk to be operated. A_{curr} is then assigned the broadcasted value of $A[i]$, while $\{X_1, X_2, ..., X_i, ..., X_z\}$ are temporary scratch variables. First, as in Line 4, X_i is assigned the value VPMADD52LUQ(X_i,A_{curr},A_i). Then, in a tight loop from $i+1$ to z, the temporary products VPMADD52LUQ(ZERO, A_{curr}, A_i) are computed, left-shifted, and added to previously computed X_j. From our discussion in Section 8.2, it is clear that the *third* input operand to IFMA instructions is vulnerable to leakage. With GCC, it is A_i that is emitted to be the third operand. This is intuitive: note that in VPMADD52LUQ(ZERO, A_{curr}, A_i), the first two operands are *fixed* for the entire duration of the loop and thus are pre-loaded by the compiler (as first and second operands of VPMADD52LUQ) in registers zmm1 and zmm3. Since A_i changes in every iteration, this becomes the *third* (memory) operand of VPMADD52LUQ. Therefore, the leaked value is the multiplicand $A_i : i \in \{i+1, i+2, ..., z\}$, which constitutes the actual input A to AMM/AMS.

Listing 2: Implementation of MulALPart algorithm [17]

```
1  .set i, 0
2  ; Initialize Xi in zmm0
3  ; Initialize Acurr in zmm1
4  ; Xi = VPMADD52LUQ Xi, Acurr, Ai
5  VPMADD52LUQ i(%rcx), %zmm0, %zmm1
6  .rept N    ; iteration bounds (i+1, z)
7    vpxord %zmm3, %zmm3, %zmm3  ; T = 0
8    ; T = VPMADD52LUQ ZERO, Acurr, Ai
9    VPMADD52LUQ i(%rcx), %zmm1, %zmm3
10   vpslld $1, %zmm3, %zmm3  ; T = T << 1
11   vpaddd %zmm1, %zmm1, %zmm3 ; Xi = Xi + T
12 .set i, i+1
13 .endr
```

```
1 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
2 c0 25 a5 a9 1a 26 90 88 fa fa fa fa fa fa fa fa
3 30 5d c3 45 fd fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
4 fa fa fa fa fa fa fa fa fa fa fa 7f f2 6c 11 30 6b fa fa fa fa fa fa fa fa
```

Listing 3: Leakage when victim executes Listing 2.

As depicted in Listing 3, we successfully observe the programmed input $A = \{A_1, A_2, ..., A_z\}$ leaking in the attacker thread. Implication-wise, this compromises the *input* to AMM/AMS. Depending on the particular use-case where Montgomery multiplication and Montgomery squaring is deployed, such leakage has the potential of compromising sensitive data. For instance, in line with Intel's recommendation [23], should IFMA-enabled Montgomery multiplication/squaring be used to accelerate traditional RSA encryption, then the input to Montgomery multiplication/squaring is the victim plaintext itself. Moreover, Montgomery ladder algorithm is often used as constant-time and side-channel protected RSA and ECC implementations. In such cases, the ability of the attacker to observe the value of the operands of the Montgomery ladder algorithm essentially enables it to distinguish between the if and else part of the control flow [3, 27], thereby leaking the secret exponent. Apart from traditional public key algorithms, other forms of cryptosystems also use such multiplication and squaring. We discuss their respective FMA/IFMA accelerations and relevant vulnerabilities next.

8.3.2 Supersingular Isogeny Diffie-Hellman

In [14], the authors present IFMA based accelerations of Cumulative Supersingular Isogeny Diffie-Hellman (CSIDH), which is a post-quantum key establishment scheme in the isogeny-based cryptosystem family. The foundations of this cryptosystem rely on supersingular elliptic curves, which again require core arithemetic operations like multiplications. Intuitively, IFMA seems the perfect choice for accelerating such implementations. The authors propose parallelized operations on the elliptic curve groups, again through optimized implementations of Montgomery multiplication.

In this implementation, we target gfp_mul_8x1w[10], which is a 8-way montgomery multiplication using VPMADD52LUQ and VPMADD52HUQ. As before, we program the inputs to this function to 0xfa. Listing 4 displays the result of the attack.

```
1  ac d4 2d 65 fa fa fa fa fa fa 6b 22 f7 92 a5 f3 97
2  fa fa 8a fc bf 89 0f fa fa fa fa fa fa fa fa ca a4 f9 e8
3  a1 60 82 fa fa fa fa fa fa fa be 78 06 51 61 c2 0a
4  fa fa fa fc fa fa 0e fa fa fa fa fa fa fa 6a fa fa fa fa fa fa fa fa
5  fa fa fa fa fa fa fa fa 9c 2c 32 69 cc bf 9c 2c 32 69
6  cc bf 0f 1d 6e b9 04 50 9b 16 83 95 50 fc f6
7  95 be d0 42 ab fa fa fa 05 fa bc a7 5b a2 b9 ba 8c d9
8  d0 83 10 e1 8d d0 42 ab c4 e4 e3 05 fa fa fa 1f e9
9  fa 9f fa c4 fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
10 fa fa fa fc fa fa 0e fa f4 fa fa fa c2 fa fa fa fa fa fa fa
```

Listing 4: Leakage when victim executes gfp_mul_8x1w.

8.3.3 Supersingular Isogeny Key Encapsulation

In [13], the authors use IFMA to accelerate SIKE, which is a post-quantum isogeny based cryptosystem (similar to SIDH in Section 8.3.2). Here too, IFMA is used to speed upbase/extension field arithmetic, point arithmetic, and isogeny computations. Since SIKE relies mainly on SIDH-friendly primes, techniques for speeding up modular reduction also apply here. Henceforth, as with Section 8.3.2, this implementation[11] is also vulnerable to the leakage demonstrated here.

A slightly different take on accelerating SIKE can be found in [32], which uses VPMADD (of the FMA family) instead of IFMA, to achieve similar aims of accelerating elliptic curve arithmetic. However, the vulnerability still stands, since VPMADD* sub-family of FMA instructions are also vulnerable

[10] https://gitlab.uni.lu/APSIA/AVX-CSIDH
[11] https://gitlab.uni.lu/APSIA/AVXSIKE

to this leak. This fact thereby reinforces the extent of the leakage's impact, irrespective of the underlying cryptosystem and its specification of ISEs used.

8.3.4 Other Use-Cases

We briefly mention other use-cases which have similar vulnerability. In [45], the authors propose IFMA based Montgomery multiplication, which again leaks the operands through the leakage described here. Likewise, in [58], Montgomery reduction is also sped up using IFMA, and is vulnerable to our attack. Note that [58] speeds up Dilithium, which is a post-quantum digital signature scheme, and has similar vulnerability as AMM in Section 8.3.1 of leaking the plaintext message being signed. Finally, recent works like [6] have also demonstrated the use of IFMA to speed up homomorphic operations as well, and is also vulnerable to the leak[12].

9 Related Works

Prior works on automated discovery of speculative vulnerabilities in processors can be broadly categorized into two classes based on their approaches for detecting leakages - ① using formal methods and ② fuzzing-based approach. Such tools have their own features and limitations and do not cover the exploratory search space that we target in this work.

Detection of Leakage through Formal Methods. Revizor [38] is a model-based tool built on the fundamentals of *contract traces* in processors and detects violation in the contracts to identify potential leakages. In order to generate contracts, Revizor uses *random instruction* sequences with use-case specific pruning - straight-line code for meltdown-type violations [21] and conditional branches for spectre-type violations [39]. As the instruction sequences are chosen randomly, both [21] and [39] consider a small subset of x86 instructions such as to not blow-up Revizor's search space. Moreover, Revizor monitors changes in the L1D cache state to detect contract violations. In contrast, Shesha does not rely on random instruction sampling and pruning from apriori knowledge of target speculative behavior. It works upon the formulation of equivalence classes to provide direction to its test case generation which allows defining a "fitness function" to maximize occurrences of bad speculation in every equivalence class. Furthermore, Shesha relies on PMCs, instead of L1D cache, to observe if bad speculation occurred. Scam-V [7, 37] is another tool that relies on model-based testing by generating instruction sequences as test-cases to record observation. Recently published Plumber [22] extends Scam-V to generate leakage templates based on changes in cache state. Both Scam-V and Plumber target ARM-ISA and uses symbolic execution to identify leakages. In contrast, Shesha targets x86-ISA and attempts to find novel leakage paths instead of matching templates.

Detection of Leakage using fuzzing. Although traditionally, fuzzing has been used to detect bugs in software, hardware-based fuzzing [16] has seen increasing popularity in recent years. *Transynther* [36] relies upon mutating known code sequences of attacks like [8, 33, 42, 50] through fuzzing to discover newer variants of Meltdown-type attacks. Similarly, Speechminer [55] generates random code snippets to detect speculation-based attacks. Both these tools use random fuzzing with templates, such as transient window enlargement gadget, faulted load, disclosure gadget, etc. that are known apriori in literature. Since, these tools were developed to target certain classes of speculations, they cannot be directly extended to explore the vast space of bad speculation. In contrast, Shesha does not place any restrictions on the explored bad speculation classes from apriori decisions. Furthermore, these tools work with limited subset of instructions and do not cover the ISEs handled by Shesha. Similarly, automation tools such as *Osiris* [52] and *Absynthe* [20] explore only contention-based attacks and not transient leakages.

10 Mitigations

We discuss purely software based mitigations against the transient paths discovered by Shesha.

Precision based assists and machine clears: As pointed out in Section 6.2, the SMC occuring due to the precision during hardware assist is managed as *loads*/*stores*. Thereby, an `lfence` before and after an instruction sequence with a homogeneous precision type (either double or single precision) *serializes* the load and eliminates transient execution.

SIMD-Vector Transition based SMC: As discussed in Section 6.1, SIMD-Vector transitions based SMC occur to effectively issue *blend* operations. As Intel Optimization Manual suggests [24], SIMD-Vector based SMC can be prevented by appropriate use of zero-latency instructions `vzeroupper`/`vzeroall` before and after a function executing SIMD or Vector instruction sequences.

Assists due to denormal arithmetic: Intel suggests to set the flags `FTZ` (Flush to Zero) and `DAZ` (Denormals are Zero) to prevent denormal numbers from occurring, thereby preventing the floating point assists. However, disabling denormals violates IEEE-754 standard and is not a viable solution for most applications and use-cases [18]. Moreover, custom compiler-level mitigations like [40] have an unacceptable overhead.

FMA leakage: One obvious mitigation is to disable Simultaneous Multi-Threading (SMT), albeit with performance hits. Another (software) mitigation strategy involves using `lfence` to serialize loads and prevent speculation, transitively preventing the leakage described in this work. An architectural change to disassociate FMA from the SIMD buffer can stop FMA-AVX cross-execution unit leakage.

[12] It is unclear, though, what the extent of exploitability is in the case of homomorphic encryption. This is because even though there is leakage, all data is essentially encrypted and thus secure by the underlying guarantees of the homomorphic encryption scheme.

11 Conclusion

Bad Speculation in modern processors causes all in-flight micro-operations to be flushed from the pipeline, contributing to transient execution paths. In this work, we develop an automated transient leakage detection tool, named Shesha, inspired by Particle Swarm Optimization principles. We establish the concept of equivalent classes that represent disjointedly fragmented sub-spaces of bad speculation, which lead to faster convergence and allows us to cover vast search space of ISEs. Using Shesha, we discover novel transient leakage paths and reverse-engineer the root-cause of such transient executions. We discuss how the uncovered speculative execution paths can be used as building blocks for micro-architectural attacks. Finally, we show leakage on cryptographic libraries that use IFMA to accelerate multiplications; we demonstrate how data from FMA execution engine is speculatively forwarded to AVX execution engine. Overall, transient executions are hard to mitigate, and with new optimizations being implemented in next-generation processors, the possibility of such transient paths only seems to be accentuated.

Acknowledgment

The authors would like to thank the reviewers and the shepherd for their suggestions in improving the paper. They would also like to thank the Department of Science and Technology (DST), Govt of India, IHUB NTIHAC Foundation, C3i Building, Indian Institute of Technology Kanpur, and Centre on Hardware-Security Entrepreneurship Research and Development, MeitY, Govt of India, for partially funding this research.

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, ASPLOS '19, pages 673–686, New York, NY, USA, 2019. ACM.

[2] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against {Cross-Privilege} spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, 2022.

[3] Daniel J Bernstein and Tanja Lange. Montgomery curves and the montgomery ladder. 2017.

[4] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 785–800, 2019.

[5] Fabian Boemer and Vinodh Gopal. Fused multiple multiplication and addition-subtraction instruction set, September 21 2023. US Patent App. 17/695,554.

[6] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, and Vinodh Gopal. Intel hexl: accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 57–62, 2021.

[7] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. Validation of side-channel models via observation refinement. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 578–591, 2021.

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.

[9] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[10] Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay. Shesha: Multi-head microarchitectural leakage discovery in new-generation intel processors. https://arxiv.org/abs/2406.06034, 2024.

[11] Anirban Chakraborty, Nikhilesh Singh, Sarani Bhattacharya, Chester Rebeiro, and Debdeep Mukhopadhyay. Timed speculative attacks exploiting store-to-load forwarding bypassing cache-based countermeasures. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 553–558, 2022.

[12] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[13] Hao Cheng, Georgios Fotiadis, Johann Groszschädl, and Peter YA Ryan. Highly vectorized sike for avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2022(2), 2022.

[14] Hao Cheng, Georgios Fotiadis, Johann Groszschädl, Peter YA Ryan, and Peter Roenne. Batching csidh group actions using avx-512. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2021(4):618–649, 2021.

[15] Jesus Corbal, Robert Valentine, Roman S Dubtsov, Nikita A Shustrov, Mark J Charney, Dennis R Bradford, Milind B Girkar, Edward T Grochowski, Thomas D Fletcher, Warren E Ferguson, et al. Systems, apparatuses, and methods for chained fused multiply add, December 4 2018. US Patent 10,146,535.

[16] Christopher Domas. Breaking the x86 isa. *Black Hat*, 1:1–6, 2017.

[17] Nir Drucker and Shay Gueron. Fast modular squaring with avx512ifma. In *16th International Conference on Information Technology-New Generations (ITNG 2019)*, pages 3–8. Springer, 2019.

[18] FOSS. A floating point error that caused a damage worth half a billion. https://itsfoss.com/a-floating-point-error-that-caused-a-damage-worth-half-a-billion/#:~:text=So%2C%20what%20exactly%20happened%3F,and%20thus%20the%20conversion%20failed, 2023.

[19] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1871–1885, 2020.

[20] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic black-box side-channel synthesis on commodity microarchitectures. In *NDSS*, 2020.

[21] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. Speculation at fault: Modeling and testing microarchitectural leakage of {CPU} exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7143–7160, 2023.

[22] Ahmad Ibrahim, Hamed Nemati, Till Schlüter, Nils Ole Tippenhauer, and Christian Rossow. Microarchitectural leakage templates and their application to cache-based side channels. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1489–1503, 2022.

[23] Intel. Intel 64 and ia-32 architectures optimization manual. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html, 2013.

[24] Intel. Intel 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2013.

[25] Intel. Gather data sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/gather-data-sampling.html, 2023.

[26] Intel. Fast modular multiplication technique guide. https://networkbuilders.intel.com/solutionslibrary/intel-avx-512-fast-modular-multiplication-technique-technology-guide, 2024.

[27] Marc Joye and Sung-Ming Yen. The montgomery powering ladder. In *International workshop on cryptographic hardware and embedded systems*, pages 291–302. Springer, 2002.

[28] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. IEEE, 1995.

[29] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.

[31] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.

[32] Dusan Kostic and Shay Gueron. Using the new vpmadd instructions for the new post quantum key encapsulation mechanism sike. In *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pages 215–218. IEEE, 2019.

[33] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.

[34] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on*

Computer and Communications Security, pages 2109–2122, 2018.

[35] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[36] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

[37] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I 32*, pages 225–248. Springer, 2020.

[38] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box cpus against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.

[39] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752. IEEE, 2023.

[40] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1451–1468, 2021.

[41] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021.

[42] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[43] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part I 24*, pages 279–299. Springer, 2019.

[44] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[45] Daisuke Takahashi. Fast multiple montgomery multiplications using intel avx-512ifma instructions. In *Computational Science and Its Applications–ICCSA 2020: 20th International Conference, Cagliari, Italy, July 1–4, 2020, Proceedings, Part V 20*, pages 655–663. Springer, 2020.

[46] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Meltdownprime and spectreprime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[47] 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 *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[48] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 54–72. IEEE, 2020.

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

[50] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105. IEEE, 2019.

[51] Stephan Van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 339–354. IEEE, 2021.

[52] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432, 2021.

[53] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. *Technical report*, 2018.

[54] Johannes Wikner and Kaveh Razavi. {RETBLEED}: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3825–3842, 2022.

[55] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. *arXiv preprint arXiv:1912.00329*, 2019.

[56] Wenjie Xiong and Jakub Szefer. Survey of transient execution attacks. *arXiv preprint arXiv:2005.13435*, 2020.

[57] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In *23rd USENIX security symposium (USENIX security 14)*, pages 719–732, 2014.

[58] Jieyu Zheng, Haoliang Zhu, Zhenyu Song, Zheng Wang, and Yunlei Zhao. Optimized vectorization implementation of crystals-dilithium. *arXiv preprint arXiv:2306.01989*, 2023.

Appendix

A Shesha: An algorithmic perspective

Algorithm 1 Shesha driver

```
1: procedure DRIVER
2:     ise_list ← construct_ise_list()
3:     pos_vector_set ← construct_population_set(ise_list)
4:     cognitive_phase(pos_vector_set, ise_list)
5:     Create sub-swarm set S from equivalence class of each member
6:     mixed_phase(pos_vector_set, ise_list, S)
```

Algorithm 2 Shesha

```
1: procedure CONSTRUCT_ISE_LIST
2:     instructions ← parse ISE xml
3:     ise_dict ← {}
4:     for inst in instructions do
5:         ops ← Parse operands for inst
6:         ise_dict[inst] ← ops
7:     return ise_dict
8: procedure CONSTRUCT_POPULATION_SET(ise_list)
9:     population_set = []
10:    for member_index in N do
11:        instruction_sequence ← []
12:        for dimension in n do
13:            instruction ← sample instruction from ise_list
14:            instruction_sequence.append(instruction)
15:        population_set.append(instruction_sequence)
16:    return population_set
17: procedure COGNITIVE_PHASE(pos_vector_set, ise_list)
18:    α = 1, β = 0.4, γ = 0
19:    iter ← choose iterations
20:    while iter is not 0 do
21:        for i in N do
22:            inst_prob ← [0, 1]
23:            if inst_prob > β then
24:                d ← random sample from {1,2,3,....,n}
25:                pos_vector_set[i][d] ← sample from ise_list uniformly at random
26:            operand_prob ← [0, 1]
27:            if operand_prob > β then
28:                d ← random sample from {1,2,3,....,n}
29:                Mutate operands of pos_vector_set[i][d]
30:            Evalute fitness and assign equivalence class to pos_vector_set[member_index]
31:        iter = iter - 1
32: procedure MIXED_PHASE(pos_vector_set, ise_list, S)
33:    α = 1, β = 0.4, γ = 0
34:    iter ← choose iterations
35:    while iter is not 0 do
36:        for i in N do
37:            S_i ← determine sub-swarm
38:            inst_prob ← [0, 1]
39:            Mutate instructions as in cognitive phase
40:            if inst_swarm_prob > γ then
41:                d ← random sample from {1,2,3,....,n}
42:                pos_vector_set[i][d] ← sample from leader of sub-swarm S_i
43:            operand_prob ← [0, 1]
44:            Mutate operands as in cognitive phase
45:            Evalute fitness and assign equivalence class to pos_vector_set[member_index]
46:        iter = iter - 1
```