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 artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears 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 Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

USENIX Security '24 Artifact Appendix: Shesha: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors

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

A Artifact Appendix

A.1 Abstract

Shesha is a transient execution finding tool for modern Intel CPUs. It leverages the principles of particle swarm optimization (PSO) to automatically generate instruction sequences to trigger novel transient leakage paths in the processor. This artifact consists of the shesha tool, the proof-of-concept attack building blocks presented in the paper and the Fused Add-Multiply based leakage attack codes. The repository also contains the `asm` instruction sequences that can trigger different transient execution events, as tested on our experimental platform Alder lake (12th Gen Intel(R) Core(TM) i7-12700 : Microcode version 0x34) and Rocket Lake (11th Gen Intel(R) Core(TM) i5-11500 : Microcode version 0x57).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The evaluation of Shesha requires installing Python packages and accessing Hardware Performance Counters (HPCs). Therefore, the user must have `sudo` access in the system to run the artifact. The attack codes and PoCs are provided as-is for the purpose of artifact evaluation and reproducing the results. The authors shall not be held responsible for any problems caused by using the provided codes for other purposes.

A.2.2 How to access

The artifact is available in GitHub repository: https://github.com/SEAL-IIT-KGP/shesha/releases/tag/v2 (latest release; stable link).

A.2.3 Hardware dependencies

To run the artifact, you need a machine with modern Intel client processors (11th, 12th, and 13th Gen) or Xeon processors (3rd and 4th Gen), running Ubuntu OS natively (not on a virtual machine). You need to enable hyper-threading for the attacks (the tool does not require hyperthreading). Additionally, about 5GB disk and 16GB RAM is recommended for smooth operation of the tool.

A.2.4 Software dependencies

The artifact is based on Python, C, asm and bash. Detailed requirements are provided in README (with the artifact).

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

The users are required to install:

- `numpy` and `datetime` (as part of `tool/requirements.txt`)
- `gcc` (v11.4.0 or greater). This is a already part of `tool/setup.sh`
- `nasm` (v2.15.05 or greater). This is already a part of `tool/setup.sh`.

A.3.2 Basic Test

The tool and the PoCs need specific instructions to be available in the ISA supported by the processor. Run the `setup.sh` to download the instructions list, required packages of Python and check required flags in the processor.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): [Shesha tool] Shesha, with its particle swarm optimization, is able to span expansive search spaces of over 500 instruction type and uncover previously undiscovered avenues of bad speculation as detailed in Table 2 of the paper. Shesha successfully mixes exploration (Section 3.4.2 of the paper) and exploitation (Section 3.4.3 of the paper) to mutate particles in order to maximize bad speculation. In about two hours, most of the speculation events (claimed in the paper) should be visible on affected CPUs.

(C2): [speculation_pocs] The events uncovered by the tool in **C1** (also detailed in Table 2 of the paper) demonstrate speculative behaviour (i.e. allow execution of instructions which are otherwise not executed architecturally), when tested across Intel 11-th and 12-th gen client-side processors, and Intel 3-rd gen and 4-th gen server-side processors. This is demonstrated by transiently executing a string that is never supposed to be executed architecturally and then leaking it using Flush+Reload channel.

(C3): [fma_data_leak] As detailed by Section 8 of the paper, the tool in **C1** uncovers a novel data leak from the FMA execution engine to the AVX execution engine. This encompasses 60+ instructions belonging to both FMA and IFMA extensions. Concretely, the memory operands to FMA/IFMA instruction extensions buffer their data in "Memory Access Units" with the FMA execution unit, and such operands are "speculatively" forwarded to faulted loads induced in the AVX execution engine. This is demonstrated by leaking data from victim process running FMA instructions, to the hyperthreaded attacker process.

A.4.2 Experiments

(E1): [Shesha tool] *[1 human-hour + 3 compute-hours + 5GB disk + 16GB RAM]*:

How to: Run `setup.sh` to download the instructions list and install the required packages. It also installs MSR kernel module for reading HPCs.

Preparation: The script `setup.sh` will automatically prepare the test platform. The user needs to check if the CPU flags used in the next instructions are available in their CPU. The terminal output of `setup.sh` shows all the available flags in the CPU.

Execution: Run the command `python3 shesha.py -num-instructions 40 -avx -avx2 -sse -ssse3 -sse2`. The user can change the argument `-num-instructions` to any non-zero positive integer value. This argument controls the number of instructions to start in that each particle manages.

Results: As Shesha starts exploring different instruction sequences for bad speculation events, the `run.log` output file stores the instances when Shesha discovers new transient events, along with the generation number. Additonally, all such instruction sequences that generated transient leakages are stored in the `asm` directory.

(E2): [AVX-SSE transient leak] *[0.25 human-hour + 0.25 compute-hour]:*

How to: The codes can be found at `shesha/speculation_pocs/simd_vector` directory. Note that this vulnerability is only present in Intel 12 and 13 gen client CPUs and 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen. In case of other affected systems, the user can change the perfmon address[1] in `msr.config`.

Preparation: These experiments require huge page tables enabled. This can be enabled using the following command: `echo 16 | sudo tee /proc/sys/vm/nr_hugepages 1>/dev/null`.

Execution: The Makefile inside the directory has two targets: `perfmon` and `leak`. The `perfmon` target demonstrates the relevant performance counters for the ASM snippet contained within `fuzz.S`. Build the targets using `make all` and then run using `sudo ./fuzz`. Likewise, run `sudo ./leak` to observe the transient leakage.

Results: `fuzz` shows the number of AVX-SSE mix microcode assist and machine clear SMC events observed. `leak` transiently executes a string "This is a test to verify if it leaks", which should not be processed architecturally. **Note**: this leakage is speculative; that is, the string should not be processed architecturally. Depending on the transient window generated, the frequency of the processor at the time of test, activity in the co-located hyperthread, and other factors, it may be possible that only a few bytes of "This is a test to verify if it leaks" leak. Therefore, even if a part of the string leaks, the speculative behaviour is still established.

Paper cross-reference: This experiment captures results from Section 7.1 of the paper.

(E3): [precision missing transient leak] *[0.25 human-hour + 0.25 compute-hour]:*

How to: The codes can be found at `shesha/speculation_pocs/precision_mixing` directory. Note that this vulnerability is only present in Intel 12 and 13 gen client CPUs and 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen.

Preparation: Same as **E2**.

Execution: Same as **E2**.

Results: Same as **E2**

(E4): [AES-SSE transient leak] *[1 human-hour + 1 compute-hour]:*

How to: The codes can be found at `shesha/speculation_pocs/aes_simd` directory. Note that this vulnerability is present in Intel 11, 12 and 13 gen client CPUs and 3rd, 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen.

Preparation: Same as **E2**.

Execution: Same as **E2**.

Results: Same as **E2**.

[1] Can be found at https://perfmon-events.intel.com/

(E5): [FMA data leak] *[0.25 human-hour + 0.25 compute-hour]:*

How to: The codes can be found at `shesha/fma_data_leak/fma_source_leak_artifact` directory. To test this vulnerability, the processor must have FMA instructions. The code for this experiment is optimized for 11th gen Intel.

Preparation: Check whether the relevant flags are present in the CPU, using the following command: `lscpu | tr ' ' '\n' | grep fma`.

Configuration: ① `CPU_VICTIM / CPU_ATTACKER` : set these to co-located hyperthreads. One can check the CPU and CORE fields using `lscpu -all -extended` and ensure that attacker and victim are running on same physical core. ② `AVX_512` : Set 1 if `AVX_512` extension is available on the machine. ③ `SILENT_MODE` : Set 1 to observe the leaked output on the terminal.

Execution: Run the following commands in order. ① `make clean`, ② `make` and then ③ `./fma_leak 0 0 4`.

Results: A sample output has been shown in the repo. A successful leakage should show the victim data (in hex). In the example shown in the artifact, is leaks victim data 0x25 from co-located thread on the same core. User can change this data by changing the value of `address_fma:` in `asm.S`.

Paper cross-reference: This experiment captures results from Section 8.2.2 of the paper.

(E6): [FMA data leak] *[0.25 human-hour + 0.25 compute-hour]:*

How to: The codes can be found at `shesha/fma_data_leak/ams52_artifact` directory. To test this vulnerability, the processor must have FMA instructions. The code for this experiment is optimized for 11th gen Intel.

Preparation: Same as **E5**.

Configuration: Same as **E5**

Execution: Same as **E5**.

Results: Same as **E5**.

Paper cross-reference: This experiment captures results from Section 8.3.1 of the paper.

A.5 Notes on Reusability

Particle Swarm Optimizer is a generic methodology suited to monotonically increasing fitness functions. Evidently, the events of bad speculation used (listed in Table 1 of the paper) are monotonically increasing performance counters.

Hence, a natural extension is to consider other events in the system (for example, Model Specific Registers) that log monotonically increasing occurrences. Shesha, then, is capable of generating ASM aiming to maximize such events. We note, however, that Shesha can only generate ASMs that trigger specific behaviour wrt. the events monitored. Whether such events are exploitable or not is manual reverse-engineering and investigation.

Another potential direction is to include x86 base instructions along with the ISEs that Shesha already supports. As described in Section 3, Shesha is capable of efficient mutations that maximize the fitness function in very expansive search spaces. As such, the huge number of instructions that come with the x86 ISA can be combined with ISEs to explore other avenues of speculation (and potential exploits).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.