USENIX Security '26 Artifact Appendix: StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine

Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact provides instructions to reproduce the findings in *StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine*. Our paper introduces StackWarp, a vulnerability stemming from a hardware bug in the AMD Zen CPUs' stack engine that allows for deterministic stack-pointer manipulation across SMT threads. We provide a fuzzer to locate the undocumented Model Specific Register (MSR) bit controlling the stack engine, a Performance Monitoring Counter (PMC) analysis suite to verify this bit's function, and a Proof-of-Concept demonstrating the vulnerability.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To prevent weaponization and due to the extreme difficulty of providing host-privileged access to SEV-enabled server hardware, this evaluation focuses on the underlying hardware bug rather than a full TEE compromise.

Evaluating this artifact involves writing to undocumented MSR bits, which can potentially lead to system freezes or instability.

A.2.2 How to access

The artifact is available on Zenodo: https://zenodo.org/records/19108605

A.2.3 Hardware dependencies

In our paper, we evaluate the sychronization bug across multiple generations of AMD Zen microarchitectures. While we expect all AMD Zen 1-5 microarchitecturse to be affected, the specific microarchitectures verified in our study include:

- AMD Ryzen 5 2500U (Zen)
- AMD Ryzen 5 3550H (Zen+)
- AMD EPYC 7252 (Zen 2)
- AMD Ryzen 7 5700G (Zen 3)
- AMD Ryzen 9 6900HX (Zen 3+)
- AMD Ryzen 7 PRO 7840U (Zen 4)
- AMD EPYC 9124 (Zen 4)
- AMD Ryzen 9 9950X (Zen 5)

A.2.4 Software dependencies

We use Linux Ubuntu 22.04 and 24.04 in our experiments. We expect the following dependencies: `cmake`, `libasmjit-dev`, `libzstd-dev`, `linux-headers-amd64`, and `msr-tools`.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

1. Clone the repository and install the dependencies:

```
sudo apt install libasmjit-dev libzstd-dev
 msr-tools cmake linux-headers-$(uname -r)
```

2. Ensure the MSR driver is loaded:

```
sudo modprobe msr
```

3. Enable the unprivileged access to performance counters:

```
echo 2 | sudo tee /sys/devices/cpu/rdpmc
```

A.3.2 Basic Test

Before running the full evaluation, execute this script to ensure the hardware interfaces are accessible:

```
chmod +x ./test.sh; ./test.sh
# Expected Result:
 [+] MSR access: SUCCESS
 [+] RDPMC userspace: SUCCESS
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): StackWarp allows for manipulation of the stack pointer on sibling thread. This is proven by experiment (E1) whose results are illustrated in Figure 1.

(C2): Flipping this bit results in measurable changes on stack instructions. This is proven by the experiments (E2) whose results are reported in table 2.

(C3): Our fuzzer finds this undocumented MSR bit that affects the architectural state of the sibling thread. This is proven by the experiment (E3) described in Section 3.3.

A.4.2 Experiments

(E1): *StackWarp PoC [5 minutes]*: A minimal PoC that shows stack pointer manipulation with the MSR bit by calling an unreachable function.
How to: Navigate to the folder `poc`.
Preparation: Compile the PoC code and find two sibling cores.

```
make
# The sibling cores share the L1 and L2 cache.
lscpu -e
```

Execution: Pin the victim to a specific core, and toggle the MSR bit on the sibling core at another terminal:

```
Core1>
taskset -c <Core1> ./unreach

# Manually interrupt upon success.
Core2>
while true; do sudo bash -c 'CUR=$(rdmsr
 0xc0011029); DISABLED=$(printf "%x" $((0x$CUR
 | 524288))); ENABLED=$(printf "%x"
 $((0x$ENABLED & ~524288))); wrmsr -p <Core2>
 0xc0011029 0x$DISABLED; sleep 1; wrmsr -p
 <Core2> 0xc0011029 0x$ENABLED'; done
```

Results: The output above confirms that the synchronization bug is present on this machine. The return address was redirected to other slots on the stack.

```
Core1>
% taskset -c <Core1> ./unreach
Unreachable!!!
```

(E2): *PMC Verification [10 human-minute + 5 compute-minute]*:
How to: Navigate to the folder `ucode_instr_num`.
Preparation: TODO.

```
# Offline the SMT first to avoid crash:
echo 0 | sudo tee
 /sys/devices/system/cpu/cpu<CORE2>/online

# Disable Op Cache to reduce noices.
sudo bash -c 'CUR=$(rdmsr 0xc0011021);
 DISOP=$(printf "%x" $((0x$CUR | 32))); wrmsr
 -p <CORE1> 0xc0011021 0x$DISOP'
```

Execution: Run the PMC analysis with the stack engine on and off.

```
taskset -c <CORE1> ./test

sudo bash -c 'CUR=$(rdmsr 0xc0011029);
 DISABLED=$(printf "%x" $((0x$CUR | 524288)));
 wrmsr -p <CORE1> 0xc0011029 0x$DISABLED'

taskset -c <CORE1> ./test
```

Results: The Delta will be printed on the terminal.

(E3): *MSR Fuzzing [1 human-hour + 1 compute-hour]:*
How to: Navigate to the folder `architectural_tests`.
Preparation: Specify one pre-defined msr list in `msr_lists.h` and build the tests.

```
# Specify the msr list
#define MSR_LIST msr_list_amd_zen4
# Build
cmake -B build && make -j$(nproc) -C build
```

Execution: Insert the MSR access kernel module before running test programs:

```
cd build
sudo insmod msr_flipper.ko
# choose one of the test programs, e.g.,
sudo ./single_instructions
```

Results: The tool generates instruction sequences. It toggles the MSR bits and compares the execution state of the tested sequence.
Warning: the stackwarp bit (`0xC0011029`, bit 19) is included in the list and may cause the system to freeze. Be advised that system hangs may persist even after this bit is commented out, as MSR modifications inherently introduce stability issues.

A.5 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/usenixsec2026/.