USENIX'23 Artifact Appendix: INCEPTION: Exposing New Attack Surfaces with Training in Transient Execution

Daniël Trujillo [†]
ETH Zurich

Johannes Wikner [†]
ETH Zurich

Kaveh Razavi
ETH Zurich

[†] Equal contribution first authors

A Artifact Appendix

A.1 Abstract

Our paper introduces the new TTE class of transient execution attacks and presents an end-to-end exploit INCEPTION. In particular, we make four major claims: 1) TTE allows manipulation of the BTB and RSB in transient execution, 2) a PHANTOMCALL allows manipulation of the RSB from an arbitrary instruction, 3) our end-to-end exploit INCEPTION leaks arbitrary kernel memory, and 4) `ibpb` overhead is between 93.1% and 239.2%. To back up these claims, this artifact reproduces experiments outlined in the paper, specifically those described in Section 8, Section 7.1, Section 7.3 and Section 9.

All experiments should be run on an AMD Zen microarchitecture. Our end-to-end exploit INCEPTION requires an Zen 1(+), Zen 2 or Zen 4 microarchitecture.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our exploit INCEPTION leaks kernel data from a user-provided address. Potentially private data located at the provided address will be printed to `stdout` and stored in a file (`data.bin`). An evaluator may choose to clear `stdout` and/or remove the output file after running this experiment.

Other than this, our experiments do not impose any security, privacy or ethical concerns.

A.2.2 How to access

The source code of INCEPTION is retrieved by cloning `https://github.com/comsec-group/inception.git`. The code for this artifact can be found under git tag `usenix-23-ae-final`.

A.2.3 Hardware dependencies

All provided code should be run on an AMD Zen microarchitectures. The end-to-end exploit works only on Zen 1(+), Zen 2 and Zen 4 microarchitectures.

A.2.4 Software dependencies

All experiments were ran on Ubuntu 22.04 LTS (Jammy Jellyfish), with a Linux kernel 5.19.0-28-generic. The following packages must be installed, available in the Ubuntu apt repository. `git build-essential clang linux-{image,headers,modules,modules-extra}-5.19.0-28-generic amd64-microcode=3.20191218.1ubuntu2, python3`.

A.2.5 Benchmarks

To evaluate `ibpb` as a mitigation, download UnixBench from `https://github.com/kdlucas/byte-unixbench` and place it under `./ibpb-eval`.

A.3 Set-up

The experiments are designed to run on bare-metal, *they will not work inside a virtualized environment*. You need an AMD processor similar to the ones we used in the paper.

A.3.1 Installation

1. Install Ubuntu 22.04
2. Install necessary dependencies (c.f. §A.2.4).
3. Boot the newly installed kernel.

A.3.2 Basic Test

Navigate to the path of the repository and run `./check.sh`. This script should show three times `PASS`. If the first line shows `PASS`, but the second or third line shows `FAIL`, all experiment but **E3** can be evaluated on your system.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): TTE allows manipulation of the BTB and RSB with code executed in transient execution.

(C2): We can manipulate the RSB from an arbitrary instruction using a PHANTOMCALL.

(C3): Our end-to-end exploit INCEPTION leaks arbitrary kernel memory.

(C4): `ibpb` can be used as mitigation against INCEPTION on Zen 1(+) and Zen 2, and has an overhead between 93.1% and 239.2%.

A.4.2 Experiments

(E1.1): [TTE of BTB] [1 minute]: this experiment executes a branch in transient execution and determines whether the state of the BTB has been changed. The experiments are described in Section 8 and depicted in Figure 9.
How to: These experiments should be carried out under `tte_btb/`.
Preparation: Install the kernel module under `kmod_ibpb`.
Execution: To build and run, run `./run_all.sh`.
Results: The script runs number of TTE tests. Lines starting with `sig_*` indicates a cache signal caused by TTE. The number is further represented within the array `rb`, where it should be significantly higher than the other numbers. The other numbers serves as indication for noise and should be low or 0.

(E1.2): [TTE of RSB] [10 minutes]: this experiment executes a branch in transient execution and determines whether the state of the RSB has been changed. The experiments are described in Section 5.2 and Section 8, and depicted in Figure 1 and Figure 9.
How to: Navigate to `./tte_rsb`.
Preparation: Before running this experiment, make sure the machine is quiescent. Find two sibling cores `CORE 1` and `CORE 2` on the target machine.
Execution: Follow the instructions in the provided `README.md`. Run `./tte_rsb.sh <CORE 1> <CORE 2> <OUTPUT DIR> <OPTIONAL CLANG ARGS>`.
Results: The output files in the `OUTPUT DIR` show the hits in the reload buffer for each return executed (one column for each return). If the RSB is uneffected by TTE, `stdout` should show a diagonal line. If this diagonal line is disturbed, entries are corrupted. If the last row of the output (`Hijacked`) shows hits, speculative return targets were hijacked using TTE.
Output files `*_16.txt` are the result of transiently executing 16 calls, and they should show a corrupted entries (disturbed diagonal line) on all AMD Zen microarchitectures. On Zen 3 and Zen 4, the output should show hijacked returns (hits in row `Hijacked`). Output files `*_32.txt` are the result of transiently executing 32 calls, and they should show hijacked returns for all AMD Zen microarchitectures. However, note that depending on the microarchitectural state, the desired number of calls do not always fit in the transient window. Therefore, output may not always show hijacked returns.

(E2): [TTE of RSB using PHANTOMCALL] [10 minutes]. This experiment shows that AMD's RSB can be manipulated with a recursive PHANTOMCALL. The experiments is described in Section 7.1 and depicted in Figure 5. The results of this artifact should resemble those shown in Figure 6. From the results produced by this experiment, it should be possible to conclude that we can hijack return instructions on all Zen microarchitecture, and that for Zen 1(+) and Zen 2 this only succeeds when a workload is running on the sibling hyperthread. However, the exact (number of) entries corrupted (and potentially returns hijacked) may differ slightly, since its dependent on various circumstances, as pointed out in the paper.
How to: Navigate to `./phantomcall/zen_1_2` when running on Zen 1(+)/Zen 2, or navigate to `./inception/zen_3_4` when running on Zen 3/Zen 4.
Preparation: Before running this experiment, make sure the machine is quiescent. Find two sibling cores `CORE 1` and `CORE 2` on the target machine.
Execution: Follow the instructions in the provided `README.md`. To start, run: `./recursive_pcall.sh {ZEN/ZEN2/ZEN3/ZEN4} <CORE 1> <CORE 2> <OUTPUT DIR> <OPTIONAL CLANG ARGS>`. As an example, if running on Zen 2, and if cores 1 and 9 are sibling hyperthreads, you may want to run: `./recursive_pcall.sh ZEN2 1 9 zen2_output`.
Results: The experiment produces up to two output files in the `OUTPUT DIR`:

1. `no_ht.txt`: this file contains the output of running the experiment on `CORE 1`, while `CORE 2` is disabled.
2. `ht.txt` (only for Zen 1, Zen + and Zen 2): this file contains the output of running the experiment on `CORE 1`, while running a workload on `CORE 2`.

The output files show the hits in the reload buffer for each return executed. The last row of the output (`Hijacked`) indicates hijacked returns (e.g. the executed return triggered the use of a transiently injected RSB entry). In case the experiment is successfull, we expect the following output:

- **no_ht**: For Zen 1(+) and Zen 2, this output should show a diagonal line which stops at a certain point, when it turns into a horizontal line. The last row (`Hijacked`) should not show hits. For Zen 3 and Zen 4, this experiment should show that we corrupt enough entries to hijack return instructions: some of the returns should show hits in the last row of the matrix (`Hijacked`).

- **ht** (only for Zen 1, Zen + and Zen 2): This output should show hits in the last row (`Hijacked`) for each column, indicating that the transient control flow was hijacked for each return executed.

(E3): [Leaking kernel memory with INCEPTION] [10 minutes]: this experiment shows that we can leak abitrary kernel memory using PHANTOMCALL, using the setup described in Section 7.3 and depicted in Figure 7.
How to: Navigate to `./inception/zen_1_2` when running on Zen 1(+) or Zen 2. Navigate to `./inception/zen_4` when running on Zen 4.
Preparation: Before running this experiment, make sure the machine is quiescent. Follow the instructions in the provided `README.md` on how to compile the required code for this experiment. Install the provided kernel module, which prints a kernel address containing a secret to `dmesg`, as described in `README.md`. When running on Zen 4, optionally enable AutoIBRS: `sudo wrmsr 0xC0000080 -a 0x200d01`.
Execution: Run INCEPTION: `./inception <KERNEL ADDRESS>`, where `KERNEL ADDRESS` can be found in the `dmesg` output.
Results: The leaked bytes are printed to `stdout`. The secret contains of 1024 As, 1024 Bs, 1024 Cs, and finally 1024 Ds.

(E4): [INCEPTION vs. `ibpb`] [10 minutes; 10 hours compute]: this experiment evaluates `ibpb` against INCEPTION.
How to: This experiment should be carried out under `./ibpb-eval`.
Preparation: Clone UnixBench, `git clone https://github.com/kdlucas/byte-unixbench`.
Execution: Run UnixBench (`./Run`) 5 times and save the results into a folder called baseline. Then reboot with the kernel parameter `retbleed=ibpb` (note: Zen3/4 requires the new kernel 6.2 parameter `retbleed=ibpb,force`) and run UnixBench 5 more times. Save the results in a folder called ibpb.
Results: To print the results in the format presented in Section 9, run `./parse.py <path>`. You may test the parser on pre-recorded results: `./parse ./raw/ee-tik-cn118`.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.