USENIX Security '25 Artifact Appendix: Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks

Alvise de Faveri Tron
Vrije Universiteit Amsterdam

Raphael Isemann
Vrije Universiteit Amsterdam

Hany Ragab
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Klaus von Gleissenthall
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

A Artifact Appendix

A.1 Abstract

We present a prototype of our pre-silicon fuzzer that uses a novel detection mechanism based on implicit secrets to find transient data leaks in CPU designs. Our artifact includes all the components of our fuzzing infrastructure, namely (1) an LLVM pass that adds bit-precise taint-tracking to the CPU simulation, (2) an AFL++ fork that adds the option to use taint as feedback, and (3) a LibAFL fuzzing driver, along with our RISC-V instruction generator and mutator, to drive the fuzzing loop. The artifact also includes the detector infrastructure we built for BOOM, namely (1) an architectural simulator that infers all architecturally-accessed locations, (2) the BOOM configuration used for fuzzing and (3) the C++ wrapper that drives the Verilated simulation and detects crashes. The artifact also includes a testsuite of minimal detectable PoCs for both known vulnerabilities (Spectre-v1, Spectre-v2, Meltdown, Spectre-v4, Spectre-RSB) and the new Spectre-LP vulnerability. Information about each component can be found in the README and in the paper.

A.2 Description & Requirements

The scripts provided in the artifact can be used to setup Phantom Trails on the BOOM core and evaluate its results.

A.2.1 Security, privacy, and ethical concerns

Experiments are completely local, and no destructive steps are taken during evaluation.

A.2.2 How to access

The main entrypoint of our artifact is available at `https://github.com/vusec/phantom-trails` at the tag `ae-initial`. The final artifact version will be available on Zenodo at `https://zenodo.org/records/14726711`.

A.2.3 Hardware dependencies

No specific hardware feature is required, however to reproduce the TTE measurements of our fuzzing campaigns it is required to run on comparable machines. Our experiments have been conducted on two machines:

1. (**M1**) AMD Ryzen Threadripper PRO 5995WX machine with 128 cores and 500 GiB of RAM
2. (**M2**) Intel Xeon Silver 4310 machine with 48 cores and 126 GiB of RAM

A.2.4 Software dependencies

No specific OS is required in principle. Our experiments have been tested on Ubuntu Linux 22.04.

A.2.5 Benchmarks

None.

A.3 Set-up

The only prerequisites are `git` and `docker`. Scripts require `bash` and `python3`.

A.3.1 Installation

Clone the repository and all dependencies:

```
git clone https://github.com/vusec/phantom-trails
cd phantom-trails
git submodule update --init --recursive
```

Build the instrumented simulation in a container (takes ≈40 minutes on a 48 cores/126GB RAM machine):

```
cd BOOM
./start.sh
```

By default, this script will build Phantom Trails with basic block coverage as feedback and the BOOM core simulation in its MediumBoom configuration inside of a container named `boom-fuzz-manual-started`. Once the script has terminated, you should get a shell inside of the docker container.

If the build script exits before completion, it is very likely that the process ran out memory during the building and linking phases of LLVM. In that case, you can modify (decrease) the number of parallel jobs by substituting `nproc` at line 21 of `BOOM/start.sh`.

A.3.2 Basic Test

After `start.sh` the shell is located in the `/chipyard/sims/verilator` folder of the Docker container. From there you can run some simple tests.

To verify that the detector works as intended, you can run:

```
phantom-trails run \
    /Samples/build/bins/pocs/meltdown-us.bin
```

This loads the selected RISC-V flat binary in the instrumented simulation's memory and runs it. The source code of the test can be found in `Samples/src/pocs/meltdown-us.S`.

The expected results is that the command will fail, and will indicate "Meltdown PF" as cause before exiting.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Phantom Trails is able to detect all known speculative vulnerabilities on the BOOM core (Spectre-v1, Spectre-v2, Spectre-RSB, Spectre-SSB and Meltdown) at the secret-access step. Experiment **E1** confirms that each PoC of our testsuite is correctly detected and classified, as reported in Section 7.1, paragraph "Poc Detection".

(C2): Phantom Trails was able to uncover a new Spectre-v1 variant, Spectre-LP, described in Section 8 of the paper. Experiment **E2** confirms that different variants of Spectre-LP, including a return-based variant and a branch-based variant, are detected by Phantom Trails.

(C3): Phantom Trails can be extended to handle MDS. Experiment **E3** reproduces the detection behavior as reported in Section 5.4 (MDS Detection).

(C4): Phantom Trails is able to find all known transient vulnerabilities on the BOOM core in reasonable time despite the lack of templates. Experiment **E4** reproduces one of our fuzzing campaigns as described in Section 7.2 (Question Q1), whose results are reported in Figure 8-10 (blue) and in Table 1 (left).

(C5): Phantom Trails is still able to perform similarly to templated approaches on SmallBoomCore. Experiment **E5** reproduces the TTE measurements reported Table 3 (rightmost column).

(C6): Phantom Trails uses simple control-flow and data-flow optimizations. Experiment **E6** described in Section 7.2 (Question Q3) shows the impact of each optimization. Our results are reported in Table 2.

(C7): Phantom Trails provides the possibility of using Taint as a fuzzing feedback. Experiment **E7** described in Section 7.2 (Question Q2) can be used to measure the impact of the taint feedback. Our results are reported in Figure 8-10 (orange) and in Table 1 (right).

A.4.2 Experiments

(E1): PoC Detection [1 compute-minutes]: Run Phantom Trails on PoCs of known vulnerabilities.
Preparation: After running `BOOM/start.sh` you should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/pocs/*;
    echo "===== Running $poc ======"
    phantom-trails run $poc;
end;
```

Results: For each sample, Phantom Trails should abort and print "Found issue: " alongside its classification. Samples should include 1 Meltdown sample (reported as `Meltdown_LOAD_PFAULT`), 2 Spectre-RSB samples, 2 Spectre-v1 samples, 1 Spectre-v2 sample, and 2 Spectre-v4 samples.

(E2): Spectre-LP [1 compute-minutes]: Run Phantom Trails on Spectre-LP snippets.
Preparation: You should be running in a `fish` shell inside of the Docker container, in the `/chipyard/sims/verilator` folder.
Execution: Run the following (`fish` syntax)

```
for poc in /Samples/build/bins/spectre-lp/*;
    echo "===== Running $poc ======"
    phantom-trails run --init=/dev/null $poc;
end;
```

Results: The folder includes 3 Spectre-LP PoCs. For each of them, Phantom Trails should abort and print "Found issue: Spectre_v1_new".

(E3): MDS [15 compute-minutes]: Build BOOM with MDS and run Phantom Trails to verify that MDS is detected.
Preparation: Inside of the Docker container, build BOOM with MDS-SB:

```
phantom-trails build --config MDSConfig -j<JOBS>
```

Execution: Run the following

```
phantom-trails run --mds --config=MDSConfig \
    /Samples/build/bins/mds-tests/mds.bin
```

Results: The command is expected to abort after printing "Found issue: MDS_STORE_BUFFER".

(E4): Fuzzing Campaign with SW Feedback [24 to 48 compute-hours]: Run a fuzzing campaign on MediumBoomCore with the Software Feedback.

Preparation: You should be inside of the Docker container. TTEs reported in Table 1 of the paper (left part) have been observed on machine **M1** described in Section A.2.3.

Execution: Run the fuzzer

```
phantom-trails fuzz
```

By default, this will run the fuzzer until all samples in `expected_findings` are found. You can kill the fuzzer at any time with:

```
sudo killall sim-fuzzer && sudo killall run-FuzzConfig
```

Results: While the fuzzer is running, you should see TTEs being updated live in the TUI (top-right panel). Once the fuzzer has found all expected crashes, the results of the campaign are available in `/chipyard/sims/verilator/out`. The same folder is also available outside of the container, under `BOOM/results/manually-started/<timestamp>/`. Crashing inputs after classification can be found in the `out/causes` folder, in flat-binary format. You can retrieve time-to-exposure and iterations-to-exposure from within the container with:

```
# Print TTEs.
/external/BOOM/eval-results-folder.py -z out/
# Printe ITEs.
/external/BOOM/eval-results-folder.py -t out/
```

(E5): Fuzzing Campaign on SmallBoom [< 1 compute-hour]: Run a fuzzing campaign on SmallBoomCore for Meltdown and Spectre-v1 (used to compare against SpecDoctor).

Preparation: Inside of the Docker container, build BOOM in the SmallBoom configuration:

```
phantom-trails build --config SmallFuzzConfig -j<JOBS>
```

Modify `expected_findings` to match SpecDoctor.

```
# Copy old list.
cp expected_findings expected_findings_all
# Create new list.
cat expected_findings_all | head -n 2 > expected_findings
# Only Spectre v1 and Meltdown should be remaining.
```

Finally, remove old results and fuzzing queue.

```
rm -rf out/*
rm .cur_input*
```

Execution: Run the fuzzer

```
phantom-trails fuzz --config SmallFuzzConfig
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: TTEs reported in Table 3 of the paper (rightmost column) have been observed on machine **M2** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

(E6): Impact of Fuzzing Optimizations [40 minutes + 24 compute-hours]: Remove optimizations from the fuzzer and verify their effect on vulnerability discovery.

Preparation: To run a fuzzing campaign without optimizations you will need to apply the patches located in `phantom-trails/eval-patches`. In principle it is possible to apply one patch at a time and re-run the fuzzing campaign. The most evident effect should be visible by removing both control- and data-flow optimizations ("Basic" column in Table 2).

```
# Exit from the container.
(container) exit
cd <PHANTOM_TRAILS_TOP_FOLDER>
# Remove dataflow opts.
git apply eval-patches/no-dataflow-*
# Remove control flow opts.
git apply eval-patches/no-cfg-*
cd Fuzzer
git apply ../eval-patches/fuzzer/no-cfg-*
```

Finally, re-build the container. This can take up to 40 minutes.

```
cd BOOM
./start.sh
```

Execution: From within the newly built container, run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.

Results: After a 24-hours run it is highly probable that only Spectre-v1 and Meltdown will be found by the tool, as reported in Table 2 of the paper ("Basic" column).

(E7): Fuzzing Campaign with Taint Feedback [40 minutes + 24h to 48 compute-hours]: Run a full fuzzing campaign on MediumBoomCore with the Taint Feedback and verify TTEs.

Preparation: To run a fuzzing campaign with the taint feedback you will need to re-build the instrumented simulation. Exit from the container and run (from the phantom-trails top folder):

```
# Discard all previous modifications.
git checkout .
# Start container with Taint feedback.
cd BOOM
./start.sh "Taint"
```

This can take up to 40 minutes.

Execution: Run the fuzzer

```
phantom-trails fuzz
```

To kill the fuzzer, same instructions as Experiment E4 apply.
Results: TTEs reported in Table 1 of the paper (right part) have been observed on machine **M1** described in Section A.2.3. Statistics of the campaign can be printed as described in Experiment E4.

A.5 Notes on Reusability

Phantom Trails can be used beyond the scope of this paper to introspect the microarchitectural behavior of RISC-V programs and evaluate different feedback metrics.

- Phantom Trails help

  ```
  phantom-trails -h
  ```

  This should show a summary of the three main commands provided by the `phantom-trails` script: `build`, which can be used to build a specific configuration of the BOOM simulation (default one is `FuzzConfig`), `run`, which can be used to run the detector on a single program, and `fuzz`, which can be used to start a fuzzing campaign.

  `phantom-trails` is a simple python script that wraps the invocation of `make`, `run-FuzzConfig` (verilated simulation entrypoint) and `sim-fuzzer` (LibAFL fuzzer) for *build*, *run* and *fuzz* respectively. The content of the script can be inspected at `BOOM/scripts/phantom-trails` (or in the `/scripts/` folder within the container).

- Compile your own sample: once inside the container, you can run the following

  ```
  # Add RISC-V toolchain to env.
  bash
  source /chipyard/env.sh
  fish

  # Add new ASM file to src.
  cd /Samples/src
  echo "nop" > test.S
  cd /Samples && make

  # Run it with Phantom trails.
  cd /sims/chipyard/verilator
  phantom-trails run /Samples/build/bins/test.bin
  ```

- Debugging and Introspection: all configurations can be built with an optional debug flag

  ```
  # Clean previous sims.
  make clean

  # Build in debug mode.
  phantom-trails build -j<PROCS> --debug
  ```

  This enables the possibility of adding additional printing flags at runtime which can be used to inspect the cycle-by-cyle state of the microarchitecture

  ```
  # Run in verbose mode.
  phantom-trails run --verbose <SAMPLE>
  # Add --logfile out.log to output to a file.

  # Print cycle-by-cycle taint and RoB info.
  phantom-trails run --report <SAMPLE>

  # Inspect specific buffers.
  PRINT_REGFILE=1 PRINT_LSQ=1 phantom-trails run <SAMPLE>
  # See boom-wrapper/src/main/resources/csrc/args.h
  # for other print flags.
  ```

- Initialization: you can change or disable the initialization snippet by providing the `-init` flag to `phantom-trails run` (`-init=/dev/null` to disable)

  ```
  phantom-trails --init=/dev/null <SAMPLE>
  ```

  This will avoid prepending the init snippet to the sample under test. To inspect the state of memory right before execution (e.g., if you want to check the exact addresses of the initialization snippet and the sample-under-test in DRAM) you can run:

  ```
  # This generates <SAMPLE>.bin.memdump
  MEMDUMP=1 phantom-trails <SAMPLE>.bin

  # Disassemble the flat binary.
  riscv64-unknown-elf-objdump                              \
              -b binary -m riscv:rv64 -M no-aliases -D \
              <SAMPLE>.bin.memdump
  ```

- Other metrics: Our Dockerfile by default builds LLVM with MSAN (more precisely, our version of MSAN – BFSAN) but in principle different sanitizers can be provided at this stage. Moreover, we build the fuzzing instrumentation using AFL++, and in principle more coverage maps can be added to the codebase we already provide.

- Other buffers: While we use the Physical Register File as a taint sink, it is possible to move the sink to other buffers as well. For this purpose, you can modify `checkTaintSinks()` in `boom-wrapper/src/main/resources/csrc/Simulator.h`.

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/usenixsec2025/.