USENIX Security '24 Artifact Appendix: InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2

Sander Wiebing* Alvise de Faveri Tron* Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

Our paper presents InSpectre Gadget, an in-depth Spectre gadget analyzer that can be used to identify viable disclosure gadgets on large codebases. We used such analyzer to uncover the residual attack surface for cross-privilege Spectre v2 in the Linux Kernel. We demonstrated the significance of such an attack surface by showing the first end-to-end Spectre v2 attack against the Linux kernel that does not require eBPF to craft gadgets (Native BHI). We also showed how the abundance of both disclosure and dispatch gadgets can be used to bypass all modern mitigations, including FineIBT. Finally, we measured the size of speculation windows for both IBT and FineIBT on a variety of microarchitectures, and the effects of SMT contention on such windows. This artifact contains all the code needed to reproduce the analysis of the Linux Kernel, as well as the PoCs for both Native BHI and the FineIBT bypass and the experiments on (Fine)IBT speculation windows.

A.2 Description & Requirements

There are four main artifacts that are relevant for this evaluation: 1) Linux Kernel analysis; 2) Native BHI; 3) Speculation-window experiments; 4) FineIBT bypass. Different requirements may apply for the different artifacts.

A.2.1 Security, privacy, and ethical concerns

Running the Linux Kernel analysis does not pose any risk, and the PoCs for both Native BHI and the FineIBT bypass are designed to only leak data locally.

A.2.2 How to access

All of our artifacts are available at the following url https://github.com/vusec/inspectre-gadget/releases/tag/v1.1, under the `experiments/` folder.

A.2.3 Hardware dependencies

There are no hard requirements for running Linux kernel analysis, however, the amount of available memory can have a minor impact on the number of reported gadgets due to execution paths being killed prematurely. For the evaluation reported in the paper, we used a 13th Gen Intel i9-13900K with 32 cores and 64GB of RAM.

For the other artifacts, the following CPUs are required for evaluation.

- **CPU: 13th Gen Intel(R) Core(TM) i9-13900K.** Required for Native-BHI, FineIBT bypass, and speculation-window experiments.
- **CPU: 12th Gen Intel(R) Core(TM) i9-12900K.** Required for speculation-window experiments.
- **CPU: 11th Gen Intel(R) Core(TM) i7-11800H.** Required for speculation-window experiments.

A.2.4 Software dependencies

The tool itself requires `python3`, and the test scripts are written in `bash`. The only requirement for the Linux Kernel analysis is to have `docker` installed. The analysis was tested on Ubuntu 23.04.

For Native-BHI, Fine-IBT Bypass and FineIBT Speculation Window the following requirements apply:

- Linux kernel 6.6.0-rc4, custom build. Source code available here: https://github.com/torvalds/linux/archive/refs/tags/v6.6-rc4.tar.gz.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To access the artifacts, first clone https://github.com/vusec/inspectre-gadget and checkout to the `v1.1` tag.

```
git clone git@github.com:vusec/inspectre-
    gadget.git
cd inspectre-gadget
git checkout v1.1
```

To run the scanner locally, you will need to install the following:

```
sudo apt-get install python3 python3-pip bat
pip3 install -r requirements.txt
```

If `bat` is not available, you can download it from https://github.com/sharkdp/bat.

Finally, you can navigate to the `experiments/` folder. Here you will find a folder for each of the artifacts to evaluate.

Linux Kernel analysis. All scripts can be found under the `scanner-eval/` folder. The only requirement is to have `docker` installed on the system.

Native BHI. All scripts are under the `native-bhi/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../native-bhi/kernel
   ./build_kernel.sh
   ```

Speculation window experiments. All scripts are under the `speculation-windows/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../speculation-windows/kernel
   ./build_kernel.sh
   ```

   Note: If your architecture is not supported by the Ubuntu config, create a config via `make localmodconfig` after you extract the Linux source code. Although we did not test it, the tests should not be dependent on a kernel configuration.

3. Please isolate two performance sibling cores by adding `isolcpus=` to the kernel boot paramters (.e.g, `isolcpus=2, 3`). Adjust the core numbers in the file `src/targets.h`. Please reboot the system and verify if the cores are isolated:

   ```
   cat /sys/devices/system/cpu/isolated
   // should print the isolated cores
   ```

FineIBT Bypass. All scripts are under the `fineibt-bypass/` folder.

1. Install dependencies:

   ```
   cd ../poc-common
   ./install_dependencies.sh
   ```

2. Build and install the kernel with the tested configuration. The required PoC patch will be applied. Next reboot into the new kernel. Note: you have to disable secure boot.

   ```
   cd ../fineibt-bypass/kernel
   ./build_kernel.sh
   ```

A.3.2 Basic Test

The `tests/test-cases` folder contains a set of simple cases to verify how the scanner behaves in various situations (cmove, branches ecc.). A single test can be ran for example as:

```
cd tests/test-cases
make
./run-single.sh used_memory_avoider
```

This should result in the scanner reporting two potential transmissions, both at address `0x4000014`. The two transmissions come from the same expression, but a different part of the expression is considered secret each time.

It is recommended to install `batcat` to visualize the resulting annotated assembly with `batcat output/asm/*`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We analyzed the Linux kernel version 6.6-rc4 (latest at time of writing) with the default configuration. We found a total of 922 and 589 exploitable gadgets in kernel indirect call and jump targets (respectively). Section 5.4 of the paper reports results of our analysis (specifically, Table 1 and Table 2). Moreover, the cumulative distributions of the gadgets found by the scanner are shown in Figure 5 and Figure 7 of the paper. Experiment (E1) reproduces the analysis on the Linux kernel and reports all the relevant numbers.

(C2): We demonstrated the Native BHI PoC can leak arbitrary kernel memory at 3.5 kB/sec on the i9-13900K CPU. This is proven by experiment (E2) which tests the leakage rate and leaks the shadow file from memory.

(C3): We show that a speculation window is present at the IBT and FineIBT defense mechanisms. We claim that we can fit up to 5, 1 and 1 dependent loads in the IBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. We claim that we can fit up to 5, 7 and 10 dependent loads in the FineIBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. This is proven by experiment (E3).

(C4): We demonstrated that the FineIBT BHI PoC can leak kernel memory at 18 B/sec on the i9-13900K CPU with a FineIBT-enabled kernel. This is proven by experiment (E4) which tests the leakage rate.

A.4.2 Experiments

(E1): [Linux Kernel analysis] [10 compute-hours + 15GB disk]: This experiment builds an image of the Linux kernel version 6.6-rc4, extracts all the call and jump targets, then runs the scanner on the kernel image and extracts a set of possible gadgets, which are then saved in a database.
Preparation: Navigate to `experiments/scanner-eval`.
Execution: Run `run.sh`. This will create a docker container, install dependencies, and automatically run the analysis (in particular, `scripts/run-eval.sh`).
Results: The results of the analysis are available in the `results` subfolder after the run. In particular, `stats.txt` contains the numbers used for tables and throughout the evaluation section, while the `figs/` subfolder contains the cumulative distribution plots. `gadgets.db` contains the database of all the gadgets, which is queried through the queries contained in `analysis/queries`.

(E2): [Native BHI] [10 compute-minutes + 15GB disk]: This experiment runs the Native BHI PoC on the custom-build kernel.
Preparation: Navigate to `native-bhi/src`. Test the working of the PoC by first skipping the huge-page finding phase:

```
sudo ./run.sh -p
```

Execution: Test the leakage rate:

```
sudo ./run.sh test_rate
```

Test and time shadow leak:

```
time sudo ./run.sh leak_shadow
```

Results: The results (leakage rate and shadow leak) are printed to the terminal.

(E3): [Speculation window experiments] [8 compute-hours + 15GB disk]: This experiment runs the speculation window experiments on the custom-build kernel.
Preparation: Navigate to `speculation-windows`. Install the kernel module:

```
cd kernel_modules/
    ibt_tests_kernel_module
sudo ./run.sh
```

Execution: Run the experiment. Replace the CPU-NAME with the CPU name (consult `lscpu`).

```
cd src
sudo ./run_test.sh CPU-NAME
```

Results: The results are available in the folder `src/results`. To analyze the results:

```
./analyze_all.sh src/results/
```

(E4): [FineIBT Bypass] [10 compute-minutes + 15GB disk]: This experiment runs the FineIBT Bypass PoC on the custom-build kernel.
Preparation: Navigate to `fineibt-bypass/src`.
Execution: Test the leakage rate:

```
sudo ./run_fast.sh
```

To test the full PoC, including the collision finding phase. Note that the collision-finding phase can take up to 5 minutes

```
sudo ./run.sh
```

Results: The leakage rate is printed to the terminal.

A.5 Notes on Reusability

While the tool has been used specifically to target Spectre-v2 in the Linux Kernel, its structure is general enough to be applicable also to other targets. Full documentation and examples of how to use the tool can be found in `docs/index.html` or at https://vusec.github.io/inspectre-gadget/.

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/.

References