A Artifact Appendix

A.1 Abstract

The artifact reproduces the results shown in Section 5 and the exploits showcased in Section 6. More specifically, we provide code to: (i) test if a system is vulnerable to BHI, (ii) verify if out-of-place BTI is possible, (iii) validate the results in Table 3, (iv) and verify the two exploits (inter- and intra- mode). The artifacts for x86-64 have been validated on Intel Core i7-10700K and Xeon Silver 4310 running Ubuntu 20.04 with Linux kernel 5.14, while the Arm results have been verified on the performance cores of a Google Pixel 6 (Cortex X1). All our source code is available on GitHub at `https://vusec.net/projects/bhi-spectre-bhb`
Following is the directory tree of the artifact:

```
bhi-spectre-bhb
├── re
│   ├── x64
│   │   ├── bhi_test          (Section 5.1)
│   │   ├── bhb_brute_force   (Section 5.2)
│   │   ├── bhb_size          (Section 5.3)
│   │   └── bhb_control       (Section 5.3)
│   └── arm
│       ├── bhi_test          (Section 5.1)
│       └── bhb_brute_force   (Section 5.2)
├── pocs
│   ├── inter_mode            (Section 6.2)
│   └── intra_mode            (Section 6.3)
└── tools
```

A.2 Artifact check-list (meta-information)

- **Experiments:** We provide self contained experiments matching the results of specific sections.
- **Compilation:** gcc, aarch64-linux-android31-clang, nasm.
- **Binary:** One binary per experiment in each directory.
- **Run-time environment:** For x86-64 experiments: Ubuntu 20.04 with Linux kernel 5.14. We provide the default Ubuntu kernel `.config` file (at the time of writing) on GitHub. For arm experiments: Android 12 with kernel 5.10. For both architectures, `bhi_test` uses a customized kernel.
- **Hardware:** x86-64 results were validated on Intel Core i7-10700K and Xeon Silver 4310. Arm results were validated on a Google Pixel 6.
- **Run-time state:** Set Linux CPUFreq governor to `performance`.
- **Execution:** Each folder contains a `./run.sh` script to run the experiment. When additional steps are required, this is specified in the README.
- **Output:** Each experiment provides only textual output. We describe in details the expected outcome for each experiment in Section A.6 and in the READMEs available in the corresponding directory.
- **How much disk space required (approximately)?:** 8GB are sufficient if the experiments are run using provided kernel images. Otherwise 80GB are needed.
- **How much time is needed to prepare workflow (approximately)?:** Few minutes in total. Each experiment and their corresponding environment can be set up with a single `./run.sh` bash script.
- **How much time is needed to complete experiments (approximately)?:** Approximately 5 minutes per experiment to run and verify the results of each experiment in `re/` and in `pocs/`.
- **Publicly available (explicitly provide evolving version reference)?:** All the source code is available at `https://vusec.net/projects/bhi-spectre-bhb`.
- **Code licenses (if publicly available)?:** Apache License 2.0.
- **Archived (explicitly provide DOI or stable reference)?:** The `ae_final` tag contains the final stable artifacts. Available at `https://github.com/vusec/bhi-spectre-bhb/releases/tag/ae_final`.

A.3 Description

A.3.1 How to access

All the source code is available at `https://github.com/vusec/bhi-spectre-bhb/releases/tag/ae_final`. Use the version under the tag *ae_final* for reproducing these results.

A.3.2 Hardware dependencies

The experiments in `re/x64/` were tested on all the Intel CPUs in Table 2. These also run on AMD, however they will not yield any interesting result since these systems are not vulnerable. The experiments in `re/arm/` were validated on a Google Pixel 6. The two end-to-end exploits (`pocs/`) were tested against the Intel Core i7-10700K and Xeon Silver 4310. Some adjustments to the cache eviction strategies and timings may be required on different Intel CPUs.

A.3.3 Software dependencies

We rely on standard build tools available in the Ubuntu package manager: `build-essentials`, `nasm`, `debootstrap` and `qemu-system-x86`. We also rely on `msr-tools` to read the `msr` specifying the availability of IBRS and eIBRS in a system. For the `bhi_test` experiments a modified Linux kernel is required. The kernel image and the source patch file are available as part of the artifact.

A.4 Installation

You can build all the artifacts from their corresponding directory using the following command depending on the target:

```
make UARCH=INTEL_10_GEN | INTEL_11_GEN | PIXEL_6
```

The only exception are `bhi_test` experiments. In order to run these, you need to first set up a VM with a custom Linux kernel (x86-64), or install a customized kernel directly (Arm). The kernel images are available as part of the artifact, as well as the patch file required to compile the kernel from source with our modifications. For x86-64, you can set up and start the VM in a few minutes following the instructions in the README found inside the `re/x64/bhi_test/vm` directory, while for Arm it is sufficient to boot the image using `fastboot boot boot.img`.

A.5 Experiment workflow

You can then execute every experiment by simply executing the `./run.sh` script in each directory.

A.6 Evaluation and expected results

In our work we make three main claims: (i) We show how Intel eIBRS and Arm CSV2 are incomplete solutions against cross-privilege BTI attacks, and introduce *Branch History Injection* (BHI) as a new primitive to build such attacks; (ii) We leverage BHI to build an end-to-end exploit on Intel systems deploying eIBRS (i.e., inter-mode); (iii) And we show that even when cross-privilege history injection is not possible kernel-to-kernel exploits (i.e., intra-mode) are still practical.

The experiments in the `re/` directory are meant to validate claim (i) for both x86-64 and Arm architectures.

The two end-to-end exploits in the `pocs/` directory are meant to validate claims (ii) and (iii).

We now describe the goal and expected output for each of these experiment. More details are available in the READMEs in each folder. The first experiments are meant to verify the claims on Intel CPUs.

- **(x64) bhi_test.**
  - *Goal.* Verify if the system is vulnerable to BHI.
  - *Implementation.* As described in Algorithm 1.
  - *Results.* On vulnerable systems we expect F+R to provide a hit rate $> 85\%$.
- **(x64) bhb_brute_force.**
  - *Goal.* Verify if we can carry out out-of-place BTI.
  - *Implementation.* As described in Figure 4, we use two different call sites and randomize the preceding jump chains.
  - *Results.* On vulnerable systems we expect stable collisions (F+R hit rate $> 85\%$) and 2^{14} iterations on average before finding a collision on Intel 10th gen CPUs—the iterations become 2^{17} for Intel 11th gen.
- **(x64) bhb_size.**
  - *Goal.* We want to recover the number of branches the BHB can keep track of.
  - *Implementation.* As described in Figure 5.
  - *Results.* We should observe predictions for $n = 29$ and $n = 66$ on the Intel Core i7-10700K and Xeon Silver 4310 respectively.
- **(x64) bhb_control.**
  - *Goal.* We want to recover the minimum number of branches under control by the attacker to generate arbitrary BTB collisions.
  - *Implementation.* As described in Figure 7.
  - *Results.* We should observe collisions for $k = 9$ and $k = 8$ on the Intel Core i7-10700K and Xeon Silver 4310 respectively.
- **(Arm) bhi_test.**
  - *Goal.* Verify if the system is vulnerable to BHI.
  - *Implementation.* As described in Algorithm 1.
  - *Results.* On the Cortex X1 we expect F+R to provide a hit rate $> 90\%$.
- **(Arm) bhb_brute_force.**
  - *Goal.* Verify if we can carry out out-of-place BTI.
  - *Implementation.* As described in Figure 4, we use same or different call sites and randomize the preceding jump chains.
  - *Results.* On the Cortex X1 we expect stable collision (F+R hit rate $> 90\%$) for in-place BTI, while no collision for out-of-place BTI.
- **(PoC) inter_mode.**
  - *Goal.* Showcase an end-to-end exploit leveraging BHI to perform cross-privilege mistraining and eBPF to read arbitrary kernel memory.
  - *Implementation.* As described in Section 6.2.
  - *Results.* It should take less than a minute to build an eviction set and then start leaking kernel memory.
- **(PoC) intra_mode.**
  - *Goal.* Showcase an intra-mode exploit where we take advantage of eBPF to perform both mistraining and misprediction.
  - *Implementation.* As described in Section 6.3.
  - *Results.* It should take less than a minute to build an eviction set and then start leaking kernel memory.

A.7 Experiment customization

Our reverse engineer programs, as well as our exploit, can also be run on different hardware with a suitable configuration. In particular, the tool `fr_checker` can be used to find the correct F+R threshold, and the file `common/targets.h` to specify the microarchitectural parameters.

A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.