USENIX'23 Artifact Appendix: BunnyHop: Exploiting the Instruction Prefetcher

Zhiyuan Zhang[†], Mingtian Tao[†], Sioli O'Connell[†],
Chitchanok Chuengsatiansup[‡], Daniel Genkin[§], Yuval Yarom[†]

[†] The University of Adelaide, [‡] The University of Melbourne, [§] Georgia Tech.

A Artifact Appendix

A.1 Abstract

We provide the artifact to demonstrate the power of BunnyHop in reverse-engineering the Instruction Prefetcher and Branch Target Buffer on Intel processors. The artifact further contains code to demonstrate BunnyHop-Reload, BunnyHop-Evict and BunnyHop-Probe in breaking KASLR and cache colored AES as well as monitoring a BTB entry cross hyper-threads.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

The evaluation of the BunnyHop-Evict involves installing a customized Linux kernel and a kernel module that does an AES encryption. To install the kernel, you may experience various *warnings* or *errors*. Please be careful when installing the kernel, and the users are on their own risk.

The provided code is only for the purpose of artifact evaluation. The authors are not responsible for any problems caused by using the provided code for other purposes.

A.2.2 How to Access

The artifact is available in GitHub repository: `https://github.com/0xADE1A1DE/BunnyHop/tree/87abca5ef855593e4dc8e40e4b162d9f01026391`.

The source code for cache colored kernel is available at `https://doi.org/10.5281/zenodo.7704477`.

A.2.3 Hardware Dependencies

To run the artifact, you need a machine with Intel processors (6th, 8th, 9th, 10th Gen), running Ubuntu OS natively (not on virtual machine). You need to enable hyper-threading.

To test the BunnyHop-Evict, a machine with Intel processor (6th $\sim$ 10th Gen) having four physical cores is necessary. Because the cache coloring we implement uses the last-level cache hash function for four core machines, we tested the BunnyHop-Evict on i7-6700 and i5-8265U.

A.2.4 Software Dependencies

You will need to install AssemblyLine and Mastik (Please refer to the README) to allocate code at any locations and use some side-channel technique APIs.

You will need essential packages to compile the customized Linux kernel. Please refer to `https://phoenixnap.com/kb/build-linux-kernel` for the full list of required packages.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

Users need to install AssemblyLine and Mastik before running any programs. You can find them under repository `https://github.com/0xADE1A1DE/AssemblyLine` and `https://github.com/0xADE1A1DE/Mastik` respectively.

Both AssemblyLine and Mastik are long term supported tools. In this artifact we use AssemblyLine available at `https://github.com/0xADE1A1DE/AssemblyLine/tree/9fb095da7b5be01a121be9262e476f7a5cf71697` and Mastik available at `https://github.com/0xADE1A1DE/Mastik/tree/8c4e550e9347e8b2f287f16f83015cd9d60414bb`.

A.3.2 Basic Test

To test if two aforementioned tools are properly installed, you can run the experiment E1.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): The instruction prefetcher prefetches multiple memory lines. (E1) proves this.

(C2): The instruction prefetcher follows trained branch. (E2) proves this.

(C3): The instruction prefetcher is shared between hyper-threads. (E3) proves this.

(C4): The branch target buffer stores branches as long and short branches and different target bits are stored. (E4) proves this.

(C5): The BunnyHop-Reload technique can be used to break KASLR. (E5) proves this.

(C6): The BunnyHop-Evict technique can be used to bypass cache coloring and table preloading. (E6) proves this.

(C7): The BunnyHop-Probe technique can be used to monitor a branch status in BTB cross-threads. (E7) proves this.

A.4.2 Experiments

(E1): [1/12 human-minutes, 1/720 CPU-hour] Test prefetching depth. For more information, please refer to README under *BunnyHop/IP_RE/test_depth*

Preparation: Have AssemblyLine and Mastik installed.

Execution: Execute the *experiment.bash* to automatically run the test. The script tests for 20 memory blocks following the invoked function.

Results: Table 1 summarizes the result collected from different platforms. On 6th $\sim$ 10th Gen processors, you should observe that the prefetch depth is 14.

(E2): [1/12 human-minutes, 1/720 CPU-hour] Test the effect of trained branches on the instruction prefetcher. For more information, please refer to README under *BunnyHop/IP_RE/test_branch*

Preparation: Have AssemblyLine and Mastik installed.

Execution: Execute the *experiment.bash* to automatically run the test. The script tests for 60 memory blocks following the invoked function.

Results: You should observe that an instruction prefetcher follows the trained branches to prefetch memory blocks. Sample result is available under the folder.

(E3): [1/12 human-minutes, 1/720 CPU-hour] Test the behavior of the instruction prefetcher on hyper-threads. For more information, please refer to README under *BunnyHop/IP_RE/test_ip_operation*

Preparation: Have AssemblyLine and Mastik installed. Set the processor governor to performance. You will need to isolate two sibling cores at the boot time. (See README)

Execution: Execute *test_idle.bash* to run the test when the hyperthread is idle. Execute *test_busy.bash* to run the test when the hyperthread is busy with fetching infinite NOPS. You will need to change pinned cores (to two sibling cores) according to your machine configuration.

Results: You should be able to plot Figure 2 on machines with 6th $\sim$ 10th Intel processors.

(E4): [1/12 human-minutes, 1/720 CPU-hour] Test the target bits stored for long branch and short branch. For more information, please refer to README under *BunnyHop/BTB_RE/test_targetbits*

Preparation: Have AssemblyLine and Mastik installed. The core runs the test is isolated at the boot time.

Execution: You will need to compile the program with the command *gcc main.c -lassemblyline -o bh*. Then you execute the program with the command *taskset -c 1 ./bh > result.txt*. In the end, you plot the graph with the command *python3 plot.py*. The graph is saved as *result.py*.

Results: You should observe that the instruction prefetcher follows the trained branches to prefetch memory blocks. The sample result is available under the folder.

(E5): [1/3 human-minutes, 1/180 CPU-hour] Break KASLR with the BunnyHop-Reload. For more information, please refer to README under *BunnyHop/bunnyhop_fr*.

Preparation: You need to find the default physical address of the target branch and branch targets. Please follow the instructions on the README.

Execution: You need to compile the program with *make*. The code we provide guesses 256 BTB tag values. To run the code, execute *bash test.bash*.

Results: You will see the obtained BTB tag bits and a computed physical address after the randomization.

(E6): [5 human-minutes, 1/12 CPU-hour] Break AES and bypassing cache coloring and table preloading with the BunnyHop-Evict. For more information, please refer to README under *BunnyHop/bunnyhop_evict*.

Preparation: You need to compile and install a kernel that supports cache coloring. You also need to install an AES kernel module. Please follow the instructions on README.

Execution: You need to first obtain the address of an AES encryption and update the value (*base*) in *test.bash* under the folder *bunnyhop_evict/self-eviction/spy*. To compile and execute the code, run the command *bash test.bash*

Results: The randomly generated plaintext and timing result are saved in file *result_0xff.txt*. To plot Figure 5, you should run the command *python3 relation.py*. It reads fewer samples and plots a Pearson correlation graph. In a scenario that the measurement is noisy, you could run *python3 process.py* to find 16 peaks.

(E7): [30 human-minutes, 1/2 CPU-hour] Test the accuracy of the BunnyHop-Probe cross hyperthreads. For more information, please refer to README under *Bunny-*

Hop/bunnyhop_pp/hyperthread.
Preparation: You need to isolate two sibling cores at the boot time. You need to update the *experiment.bash* to pin the victim and spy on two sibling threads.
Execution: The experiment is similar to that of the Flush+Reload, and it requires the attacker to find a proper waiting cycles. The waiting cycles are determined by processors and CPU frequencies. More instructions on how to find proper waiting cycles are available at the README file.
Results: The result is written to *overall_result.txt* which indicates the bits that are correctly guessed. You can get an overall success rate with the command *python3 final_analyse.py*

A.5 Notes on Reusability

We provide the template code to generate aliased branches or NOPs in *BunnyHop/src*. They can be easily adapted for different purposes. We will later integrate the BunnyHop into Mastik.

A.6 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/`.