BeeBox: Hardening BPF against Transient Execution Attacks

Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis, *Brown University*

https://www.usenix.org/conference/usenixsecurity24/presentation/jin-di

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

USENIX Security '24 Artifact Appendix

BeeBox: Hardening BPF against Transient Execution Attacks

Di Jin
Brown University

Alexander J. Gaidis
Brown University

Vasileios P. Kemerlis
Brown University

A Artifact Appendix

A.1 Abstract

This is the artifact appendix for BeeBox: a new security architecture that hardens BPF against transient execution attacks. This appendix contains instructions about how to setup, run, and reproduce the results of BeeBox, along with information regarding system and resource requirements.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact consists of scripts for setting up QEMU/KVM virtual machines (VMs) for reproducing the main experiments of BeeBox. The majority of operations on the host are unprivileged, except for a handful required to create a Debian Linux root file system that is shared across the VMs. These privileged operations are all contained in `syzkaller`'s script, `create-image.sh`. We recommend enabling password-less `sudo` while running the script to streamline its execution.

A.2.2 How to access

Our artifact is publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-ae
- Stable commit: be43784928ba43f0

The BeeBox Linux kernel is also publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-linux
- Stable commit: 29e4d7de943cb43c

A.2.3 Hardware dependencies

The current prototype of BeeBox requires a machine equipped with a 64-bit x86 processor and at least 32GB of storage space. Additionally, while not a hard requirement, we recommend the machine has at least 4 CPU cores and 8GB of RAM.

A.2.4 Software dependencies

The artifact infrastructure depends on a three main packages:

- QEMU (KVM accelerated): for virtualizing the testing and benchmarking environment.
- SSH: for controlling the running VMs.
- Python3: for various scripting tasks.

On Debian GNU/Linux, these packages can be installed with:

```
$ sudo apt-get install qemu-system-x86 \
    openssh-client python3
```

Additionally, Python packages `numpy` and `pyparsing` are required for summarizing benchmark results; these can be installed by running the following in the root of the artifact repository:

```
$ pip install -r requirements.txt
```

To check that KVM acceleration is available, the `cpu-checker` package can be installed and the `kvm-ok` program should be executed, as follows:

```
$ sudo apt-get install cpu-checker
$ sudo kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

When building the kernel variants from source the following build dependencies need to be satisfied as well (on Debian GNU/Linux):

```
$ sudo apt-get install build-essential bc kmod \
    cpio flex libncurses5-dev libelf-dev \
    libssl-dev dwarves bison
```

All other software dependencies—e.g., the BeeBox Linux kernel, Katran, `memtier`, `sysfilter`, Nginx, and Redis—are either included in the artifact repository as (stable) Git submodules or installed during the setup phase of the root file system. In both cases, everything is handled automatically and no additional work is required.

A.2.5 Benchmarks

This artifact provides three synthetic exploits and five benchmarks, corresponding to those found in Section 7 of the paper. In particular:

- `exploit`: a set synthetic programs demonstrating the effectiveness of BeeBox (§7.1.2); source code at `bpf_test/defense_effectiveness`.

- `micro`: a set of microbenchmarks (§7.2.1); source code at `bpf_test/micro_benchmark`.
- `katran`: a real-world eBPF benchmark (§7.2.2); source code at `katran`.
- `filter`: a packet filtering benchmark (§7.2.2); source code at `bpf_test/cbpf_socket_benchmark`.
- `nginx`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/nginx_test`.
- `redis`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/redis_test`.

A.3 Set-up

To reproduce the paper's major claims, we recreated the bare-metal benchmarking environment used in the paper with virtualization (i.e., QEMU/KVM). At a high-level, we first build four custom Linux kernels, namely:

- `vanilla`: a stock Linux kernel with some Spectre-PHT defenses disabled to allow Katran to run. Note that experiments with the LPM also use this kernel, but with defenses turned on.
- `hardened`: an implementation of BeeBox-RC.
- `optimized`: an implementation of BeeBox-CP and BeeBox-RB for socket filtering and XDP, respectively.
- `synthetic`: an exploit showcase; modifies the kernel to simulate an attacker's side-channel capabilities.

Then, we create a single root file system (shared across all VMs) that contains all of the benchmarks and scripts required to evaluate BeeBox. Finally, we virtualize these components to get an environment to reproduce BeeBox's results.

A.3.1 Installation

Prebuilt. To simplify evaluating BeeBox, we provide a prebuilt benchmarking environment that can be used as is (i.e., no building and installing). To use it, simply download the respective archive from Zenodo (https://zenodo.org/records/12212612/files/prebuilt.tar.gz) and decompress it. For example:

```
$ wget https://zenodo.org/records/12212612/\
files/prebuilt.tar.gz
$ tar -xzf prebuilt.tar.gz
```

Note that the prebuilt environment contains the entirety of the artifact repository, so if this option is chosen, there is no need to clone the `beebox-ae` repository from GitLab.

From scratch. To build the three kernels from scratch, enter the root of the artifact repository and run:

```
$ ./scripts/build_kernels.sh
```

Then, to build the root file system from scratch, run:

```
$ ./scripts/builds_rootfs.sh
```

After running these two commands, a root file system image and four built kernels will be found in the `build` directory. The installation process requires roughly 20GB of disk space; while compilation times vary by machine, an estimate based on our machine (8-core, 3.7GHz Intel Xeon W-2145 CPU with 64GB of DDR4 RAM) is roughly 1–2 hours.

A.3.2 Basic test

After the installation is complete, a shell to a VM running a stock kernel (i.e., the `vanilla` kernel with no hardening) can be obtained via:

```
$ ./scripts/run.sh vanilla
```

Successfully entering the VM with this command will ensure that the environment is setup correctly. The `vanilla` kernel configuration can also be swapped out for alternate configurations—namely, `lpm`, `hardened`, and `optimized`—to perform a more thorough "basic" test. Changes to the VMs do not persist across invocations (QEMU is invoked with `-snapshot`), so feel free to poke around!

A.4 Evaluation workflow

We have automated the evaluation workflow via the script `scripts/run.sh` (see its "help" menu for a complete summary of options). While the experiment descriptions below (§A.4.2) detail how to use this script to run each experiment individually, all experiments can also be batched together and run automatically via:

```
$ ./scripts/run.sh everything
```

This should take roughly 15–20 minutes to complete, producing results in both standard output and the `results` directory. At the end, it will also run `script/summary.py` to pretty-print tables summarizing the results.

Please note that while we use QEMU/KVM to recreate the environment used in the paper, the benchmark numbers presented in Section 7 of the paper were collected on bare-metal. As a result, there might be slight discrepancies between the reproduced results and those in the paper; however, overall trends should remain consistent.

A.4.1 Major Claims

(C1): *BeeBox mitigates speculative execution attacks launched from BPF programs. This is demonstrated by the experiment (E1) described in Section 7.1.2, whose results are illustrated in Table 1. In particular, BeeBox mitigates Spectre-PHT in BPF code and helpers, as well as Spectre-STL in BPF code.*

(C2): *BeeBox is more performant than the LPM for stack load and store operations and Spectre-STL. This is shown by the microbenchmarks in the experiment (E2) described in Section 7.2.1 with results shown in Figure 4.*

(C3): *BeeBox is more performant than the LPM in real-world eBPF programs. This is shown by benchmarking Katran's load balancer in the experiment (E3) described in Section 7.2.2, whose results are shown in Figure 5.*

(C4): *BeeBox exhibits $<$ 1% throughput degradation in end-to-end, real-world settings that involve packet filtering and `seccomp-BPF`. This is shown by the experiments (E4 and E5) described in Section 7.2.2, whose results are illustrated in Table 2.*

A.4.2 Experiments

(E1): *[Exploit Mitigation] [5 human-minutes + $\approx$ 0 compute-hours + $<$ 1GB disk]: demonstrates the defense effectiveness of BeeBox by showing it stops three synthetic exploits making speculative, out-of-bound accesses.*

How to: This experiment demonstrates the effectiveness of BeeBox via three synthetic exploits that run on the `synthetic` and `optimized` kernel configurations against no defenses, the LPM, and BeeBox. The exploits rely on a custom kernel module that simulates an attacker making speculative out-of-bounds accesses. This experiment corresponds to the description presented in Section 7.1.2. The source code for the test can be found in `bpf_test/defense_effectiveness`, which should already be copied in the root file system image.

Preparation: None.

Execution: First, run all exploits on an undefended kernel. To do this, enter the `synthetic` kernel with exploits initialized by running the following:

```
$ ./scripts/run.sh synthetic exploit
```

This will drop you into a shell. Verify that the exploit-helper kernel module, named `ctest`, is loaded:

```
(vm)$ lsmod
Module                  Size  Used by
ctest                  16384  0
```

Then, enter the `bpf_test/defense_effectiveness` directory and run the three exploits without any defenses:

```
(vm)$ sudo ./pht_exp
(vm)$ sudo ./stl_exp
(vm)$ sudo ./pht_helper_exp
```

Save the output of these three commands, and then run them again without `sudo` to enable the LPM defenses:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands, and `poweroff` the VM. Next, run the exploits against a BeeBox-hardened kernel by booting into a BeeBox VM:

```
$ ./scripts/run.sh optimized exploit
```

As before, check that the kernel module is installed, and then navigate to the `bpf_test/defense_effectiveness` directory and run the exploits:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands.

Result: When a synthetic exploit succeeds (i.e., defenses fail), it means that out-of-bounds memory is accessed speculatively, which is determined by timing the reload of the memory. The corresponding output should look like:

```
(vm)$ sudo ./pht_exp
[+] reload takes 64 cycles,
        in-cache reload takes 66 cycles
[+] Speculative out-of-bound access succees!
```

If a defense successfully blocks an exploit, the corresponding output should look like:

```
(vm)$ ./pht_exp
[+] reload takes 318 cycles,
        in-cache reload takes 56 cycles
[-] Speculative out-of-bound access fail!
```

An undefended kernel fails to block all three exploits; against LPM only `pht_helper_exp` succeeds; and against BeeBox all exploits are defeated. Note that the Spectre-PHT attacks have a high probability of success, while Spectre-STL attacks may need to run multiple times to succeed. To get more consistent results, try to run the experiments multiple times. For example:

```
(vm)$ for i in {1..100}; do sudo ./stl_exp; \
done | grep -q "succeed" && \
echo "succeed" || echo "fail"
```

(E2): *[Microbenchmarks] [1 human-minute + 0.1 compute-hour + $<$ 1GB disk]: run a suite of microbenchmarks across four kernel configurations. Expect the LPM overhead for the `stack` benchmark to be more than* 250%.

How to: This experiment runs a microbenchmark across the vanilla, LPM, BeeBox-RC, and BeeBox-CP kernel configurations, corresponding to the description in Section 7.2.1 and results presented in Figure 4. The source code for the benchmarks can be found in `bpf_test/micro_benchmark`, which should already be copied into the root file system image.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.

Execution: Select the `micro` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla micro
$ ./scripts/run.sh lpm micro
$ ./scripts/run.sh hardened micro
$ ./scripts/run.sh optimized micro
```

Each command will print raw results to standard output.

Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py micro
```

The results should show that LPM for the `stack` benchmark has significant overhead ($>$ 250% in our testing),

much higher than the other schemes, while BeeBox's overhead is higher in other benchmarks, but of smaller magnitude. Further, `optimized` kernel configuration (BeeBox-CP) should have less (average) overhead when compared to the `hardened` configuration (BeeBox-RC).

(E3): *[Katran Benchmark] [*1 *human-minute* + 0.1 *compute-hour* + $<$ 1*GB disk]: benchmark Katran's load balancer across four kernel configurations. Expect the LPM configuration to exhibit higher than* 100% *overhead, the hardened configuration around* 50–80% *overhead, and the optimized configuration exhibits* 15–30% *overhead.*

How to: This experiment benchmarks Katran's load balancer XDP eBPF program across the vanilla, LPM, BeeBox-RC, and BeeBox-RB kernel configurations, corresponding to the description in Section 7.2.2 and results presented in Figure 5. The source code for the benchmarks can be found in `katran`, which should already be copied into the root file system image and built.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.

Execution: Select the `katran` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla katran
$ ./scripts/run.sh lpm katran
$ ./scripts/run.sh hardened katran
$ ./scripts/run.sh optimized katran
```

Each command will print raw results to standard output.

Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py katran
```

The results should show that the LPM configuration incurs around 100% overhead, the `hardened` configuration incurs (BeeBox-RC) around 50% overhead, and the `optimized` configuration (BeeBox-RB) exhibits less than 20% overhead.

(E4): *[Filter Benchmark] [*1 *human-minute* + 0.2 *compute-hour* + $<$ 1*GB disk]: benchmark the performance of raw socket filtering using cBPF with the BeeBox-CP scheme. Expect to see overhead* $<$ 1%.

How to: This experiment benchmarks the performance of raw socket filtering using cBPF for the BeeBox-CP scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2a. The source code for this experiment can be found in `bpf_test/cbpf_socket_benchmark`, which should already be copied into the root file system.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.

Execution: Select the `filter` test option of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla filter
$ ./scripts/run.sh optimized filter
```

Each command will print raw results to standard output.

Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py filter
```

The results should show that the `optimized` version of BeeBox (BeeBox-CP) incurs $<$ 1% overhead across the benchmark programs.

(E5): *[Seccomp-BPF Benchmark] [*1 *human-minutes* + 0.3 *compute-hours* + $<$ 1*GB disk]: benchmark BeeBox's performance impact on `seccomp-BPF` for Nginx and Redis. Expect the throughput degradation of both applications to be* $<$ 1%.

How to: This experiment benchmarks the performance of syscall filtering with `seccomp-BPF` across Nginx and Redis for the `optimized` BeeBox scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2b. The source code for this experiment can be found in `bpf_test/seccomp_benchmark`, which should already be copied into the root file system.

Preparation: Enter the repository root and ensure that the host machine is sufficiently quieted.

Execution: Select the `nginx` and `redis` test options of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla nginx
$ ./scripts/run.sh optimized nginx
$ ./scripts/run.sh vanilla redis
$ ./scripts/run.sh optimized redis
```

Each command will print raw results to standard output.

Results: To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py seccomp
```

The results should show that the `optimized` version of BeeBox incurs $<$ 1% throughput degradation across the benchmark programs.

A.5 Notes on Reusability

To drop into a shell in one of the running kernels, the following command can be used:

```
$ ./scripts/run.sh [config] shell
```

where `config` is one of '`vanilla`', '`lpm`', '`hardened`', '`optimized`', or '`synthetic`'.

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/.