A Artifact Appendix

A.1 Abstract

The artifact provided with this paper comprises a benchmarking suite to evaluate the performance of booting guest kernels with Firecracker VMM modified to support in-monitor (FG)KASLR, as well as the data/scripts used to generate figures used in the paper. We leverage *perf* (Linux profiling with performance counters), and small patches to the Linux kernel to issue I/O writes to a unique port that are traced as KVM events by *perf*[1]. Benchmarking begins when Firecracker is executed, timestamps are taken before and after relevant function calls/code blocks (e.g., decompression, (FG)KASLR functionality, loading kernel segments, etc.), and the final timestamp is taken after the call to execute the guest's `init` process.

A.2 Description & Requirements

A.2.1 How to access. Artifacts can be accessed via: https://github.com/bencw12/in-monitor-rando-benchmarking

A.2.2 Hardware dependencies. Firecracker requires either Intel x86_64, or AMD x86_64, CPUs that offer hardware virtualization support. All experiments for the paper were run on a machine with an Intel Core i7-4790 CPU @ 3.60 GHz.

A.2.3 Software dependencies. Currently, Firecracker recommends either Linux kernel version 4.14 or 5.10, as those are the versions they currently use to validate source code. We ran all experiments on a machine running Ubuntu 18.04 using a Linux 4.15 kernel.

A.2.4 Benchmarks. All guest kernels, file systems, and relocation information needed to boot VMs with and without our modifications to Firecracker are included in the artifact repository. The data collected for our experiments is in the `results-paper` directory, with subdirectories containing the results for each experiment, and the included scripts will generate the graphs shown in the paper.

A.3 Set-up

Firecracker requires KVM access which can be granted with: `sudo setfacl -m u:$USER:rw /dev/kvm`. All scripts are designed to be run from a standard Linux shell with root permissions with no additional set-up.

A.4 Evaluation workflow

A.4.1 Major Claims.

- (C1): When kernels are not warm in the cache, a compressed `bzImage` achieves optimal performance due to the image being smaller than an uncompressed image, but when kernels are cached, the increase in I/O time to load an uncompressed kernel over that of a `bzImage` is small compared to the overhead incurred by the `bzImage`'s bootstrap loader. This is shown in the experiment (E2) described in Section 2.2 with results shown in Figure 4.
- (C2): The majority of the extra overhead from a `bzImage` bootstrap loader stems from decompression, which is why microVMs have moved toward directly booting uncompressed kernels. The data supporting this is also generated from (E2), and results are shown in Figure 5.
- (C3): Optimizing the `bzImage` bootstrap loader to remove decompression and redundant kernel relocations still leaves performance on the table and does not justify booting a `bzImage` over an uncompressed kernel. This experiment (E3) is described in Section 3.3 with results shown in Figure 6.
- (C4): In-monitor randomization achieves up 22% to better performance than existing/optimized methods of self-randomization where a bootstrap loader, rather than the monitor, is the controlling principle. On average, in-monitor KASLR adds a small overhead of 4% (2ms) compared to stock Firecracker. This is shown in the experiment (E4) described in Section 5.2. Results are illustrated in Figure 9.
- (C5): In-monitor randomization does not affect kernel performance outside of boot. The experiments (E5) described in Section 5.4 verify this and results are shown in Figure 10.

A.4.2 Experiments. All kernels, file systems, relocation information, and binaries are included with our artifacts, so all experiments except for (E5) can be run by executing one shell script from the root of the repository with no additional preparation. All guest kernels are Linux version 5.11, since this is the version FG-KASLR was originally patched into. Each VM is allocated 256M and 1 CPU, and the cache is warmed by booting each kernel 5 times before recording data unless otherwise specified. Each experiment finishes all 100 boots of a kernel before moving on to the next. All new data is saved in a directory separate from the data used in the paper, and will be used instead of our results by graph generation scripts if present.

Experiment (E1): *Compression Bakeoff* [1.5 compute-hours]: A comparison of overall boot times for `bzImages` compressed with six different compression schemes supported by Linux.

[1]The idea to use *perf* to trace I/O writes was found here: https://github.com/stefano-garzarella/qemu-boot-time

[Execution] Executing `run_compression_bakeoff.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/compression-bakeoff/` for each kernel during execution. To use the new data to generate a graph like Figure 3, run `scripts/fig-3.py`. LZ4 is expected to have the lowest overhead.

Experiment (E2): *Cache-Effects* [1 compute-hour]: An experiment used to demonstrate the effects of caching on overall boot time when booting a `bzImage` versus an uncompressed kernel.

[Execution] Executing `run_cache_effects.sh 100` will boot each kernel 100 times to replicate the results used in the paper. First each kernel is allowed to be warm in the cache, then each kernel is run after dropping the caches (pagecache, dentries, and inodes) to see the affect of a cold cache on boot performance.

[Results] Results are collected and saved to the directory `results/cache-effects/` automatically for each kernel during execution. The results from this experiment are used to generate Figures 4 and 5. To use the new data to generate them, run `scripts/fig-4.py` and `scripts/fig-5.py`. Figure 4 is expected to show that `bzImage`s will have faster boot times than uncompressed kernels when the cache is cold, but uncompressed kernels boot faster than `bzImage`s when they can be cached. Figure 5 is expected to show that decompression makes up the majority of bootstrapping time.

Experiment (E3): *Bootstrap Method Comparison* [1 compute-hour]: A comparison of four methods of bootstrapping Linux: *none, lz4, none-optimized,* and *uncompressed*. *none* kernels are patched to simply leave the kernel uncompressed when linking into a `bzImage`, *lz4* is an unmodified `bzImage` using LZ4 compression, *none-optimized* kernels remove decompression and extra relocations, and *uncompressed* is the uncompressed kernel natively supported by Firecracker.

[Execution] Executing `run_bootstrap_comparison.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/bootstrap-comparison/` for each kernel during execution. To use the new data to generate a graph like Figure 6, run `scripts/fig-6.py`. *none* kernels are expected to have the highest overhead, followed by *lz4*, *none-optimized*, and *uncompressed* with the lowest overhead.

Experiment (E4): *Evaluation* [2.5 compute-hours]: This experiment evaluated the performance of in-monitor (FG)KASLR by comparing in-monitor randomization with uncompressed kernels to self-randomization methods using *none-optimized* and LZ4. Each kenrel is also compared against its unrandomized counterpart as a baseline.

[Execution] Executing `run_eval.sh 100` will boot each kernel 100 times to replicate the results used in the paper.

[Results] Results are collected and saved automatically to the directory `results/evaluation/` for each kernel during execution. To use the new data to generate a graph like Figure 9, run `scripts/fig-9.py`. In-monitor randomization with uncompressed kernels is expected to have the lowest overhead compared to kernels with *none-optimized* and LZ4. Firecracker with in-monitor KASLR is expected to exhibit minimal overhead compared to stock Firecracker.

Experiment (E5): *LEBench* [5 human-minutes, 75 compute-minutes]: This experiment uses `LEBench`[2] to evaluate the performance of important kernel functions for an unrandomized kernel, and kernels with (FG)KASLR.

[Execution] Executing `run_lebench.sh <nokaslr/kaslr/fgkaslr>` will boot either an unrandomized kernel (`nokaslr`), a kernel with KASLR (`kaslr`), or a kernel with FG-KASLR (`fgkaslr`). All three kernel variants need to be run to generate the data sufficient to recreate Figure 10. Once a kernel boots, log in as `root` (username `root`, password `root`), then execute `/LEBench/run.sh`. This will run the LEBench program and the kernel will shutdown when it is finished. Repeat this process for `kaslr` and `fgkaslr`.

[Results] Results are collected and saved automatically to the directory `results/lebench/` after LEBench finishes for each kernel. To use the new data to generate a graph like Figure 10, run `scripts/fig-10.py`. The performance of kernels with in-monitor (FG)KASLR for each kernel function is not expected to deviate significantly from the baseline of `nokaslr`.

A.5 Notes on Reusability

The methods we used to benchmark the performance of the Linux bootstrap process can be extended to any part of the kernel by defining more tracepoints and placing I/O writes in the kernel code.

[2] https://github.com/LinuxPerfStudy/LEBench