USENIX Security '25 Artifact Appendix: Approximation Enforced Execution of Untrusted Linux Kernel Extensions

Hao Sun
ETH Zurich

Zhendong Su
ETH Zurich

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper on Approximation-Enforced Execution (AEE), a novel concept for ensuring the safe execution of untrusted kernel extensions even in the potential presence of the verifier's soundness bugs. The artifact consists of Linux kernel patches for version 6.7, which implement the offset, size, and tag enforcement, alongside a suite of benchmarks, build scripts, and analysis tools. This appendix provides a detailed roadmap for building the customized kernels, executing functional and performance evaluations, and reproducing the paper's results, including AEE's impact on runtime overhead, binary size, verification time, and the trusted code base (TCB) reduction.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact operates exclusively within QEMU ARM64 virtual machines (VMs) running modified Linux kernels. All evaluations involving untrusted eBPF programs are conducted in isolated VMs, ensuring that no interactions with the host environment occur. Evaluators must refrain from executing modified kernels or any associated exploits outside the QEMU environment. The artifact does not collect or transmit any data, nor does it require or expose sensitive information.

A.2.2 How to access

The complete artifact package—including kernel patches, benchmark workloads, build and execution scripts, pre-built kernel binaries, and disk images—is archived at: `https://doi.org/10.5281/zenodo.15609051`. Evaluators are advised to begin from the root directory of the unpacked archive, which contains all resources necessary for reproduction and evaluation.

A.2.3 Hardware dependencies

The artifact has been tested under the following hardware conditions:

- **CPU:** Host system capable of running QEMU with ARM64 emulation and Pointer Authentication Code (PAC) support (validated on Apple M1).
- **Memory:** At least 16 GB of RAM is recommended to ensure reliable kernel compilation and VM execution.
- **Storage:** A minimum of 50 GB of available disk space is required to accommodate kernel sources, build artifacts, disk images, and experimental results.

A.2.4 Software dependencies

The following software dependencies are required for full functionality and building relevant components:

- **Operating System:** A Linux host system is required to build the customized kernel (validated on Ubuntu 22.04). MacOS systems with Apple Silicon (M1+) are tested for running all the benchmarks in the virtual machine; other systems running on ARM64 are also feasible.
- **Emulator:** QEMU version 7.0.0 or later with ARM64 support (*e.g.*, `qemu-system-aarch64`); tested on QEMU version 9.2.2.
- **Build Tools:** `make`, `flex`, `bison`, `libssl-dev`, `libelf-dev`, `libncurses-dev`, `dwarves` for building the Linux kernel from source.
- **Cross Compiler:** `gcc-aarch64-linux-gnu`; tested with version 11.4.
- **Analysis Tools:** `Python3` and `Rust` for parsing and executing the evaluation scripts.

If the evaluator opts to use the pre-built kernels included in the archive, kernel compilation is not required, and the Linux host prerequisite may be skipped. On Ubuntu/Debian-based systems, the required dependencies can be installed with the following command:

```
apt install build-essential gcc-aarch64-linux-gnu flex
    bison libssl-dev libelf-dev libncurses-dev dwarves
    python3 rustc cargo qemu-system-arm
```

A.2.5 Benchmarks

The artifact includes four benchmark suites that collectively evaluate AEE's effects on program size, verification time, and execution time across various enforcement configurations. All benchmarks are pre-installed in the provided disk image: (1) `linux-progs`: eBPF programs extracted from Linux kernel selftests and samples; (2) `linux-bench`: benchmark programs crafted to stress specific eBPF instruction patterns and corner cases; (3) `katran`: real-world eBPF workloads from the production-level load balancer; and (4)`filter`: CPU-intensive eBPF packet filters used for performance stress testing. These benchmarks are used to quantify the potential overhead introduced by AEE's enforcement.

A.3 Set-up

A.3.1 Installation

All setup steps assume that the evaluator is operating from the root directory of the unpacked artifact archive. The artifact provides automated scripts to minimize manual intervention.

Step 1: Download Linux v6.7 Source.

```
mkdir -p build kernels
wget -P build <download_link>/linux-6.7.tar.gz
```

Step 2: Build Kernel Variants. The following script performs various tasks: (1) it unpacks the Linux 6.7 source tree; (2) it applies AEE kernel patches; and (3) it compiles five kernel variants. The kernel variants include: (1) `clean`: unmodified baseline kernel (no AEE); (2) `all`: kernel with all three AEE mechanisms enabled; (3) `offset`: kernel with only the offset enforcement; (4) `tag`: kernel with only the tag enforcement; and (5) `size`: kernel with only the size enforcement. Note that kernel compilation must be performed on a Linux host. macOS is not supported for this step due to toolchain and compatibility limitations.

```
./scripts/build_kernels.sh full
```

Step 3: Synchronize Kernel Images. After successful compilation, synchronize the kernel images to the designated directory. All resulting images will be available in `./kernels/`:

```
./scripts/sync_kernels.sh
```

Step 4: Disk Image Preparation. A pre-built ARM64 Debian Bookworm disk image is provided in `imgs/`. It includes: (1) pre-installed SSH keys for headless login; and (2) benchmark scripts and execution tools. No additional configuration is required unless re-generating the image from scratch.

A.3.2 Basic Test

To boot the baseline kernel in QEMU:

```
./scripts/boot.sh clean
```

This launches QEMU with the `clean` kernel. You should observe Linux boot messages followed by a login prompt. To login via SSH:

```
ssh -i imgs/bookworm.id_rsa -p 10023 root@localhost
```

To exit the VM safely, execute:

```
shutdown now
```

You may also test other kernel variants:

```
./scripts/boot.sh all
./scripts/boot.sh offset
./scripts/boot.sh tag
./scripts/boot.sh size
```

A.4 Evaluation workflow

A.4.1 Major Claims

The artifact supports the following key claims made in the paper:

(C1): AEE reduces the verifier's trusted code base (TCB) by 4.5x (see Table 1).

(C2): AEE prevents kernel exploitation via verifier soundness bugs (see §6.1 and Table 2).

(C3): AEE incurs 1.2% runtime and 4.8% binary size overhead on average (see Tables 3–5).

A.4.2 Experiments

Each experiment below corresponds to one or more paper claims. All experiments are self-contained and can be run using the provided scripts and disk image.

(E1) TCB Reduction (15 min manual effort). This experiment measures the reduction in the trusted code base achieved by enforcing the approximation produced by the verifier with AEE. The file `scripts/TCB_stats.pdf` reports the lines of code (LOC) for: (1) `V.a`: approximation logic in the verifier; (2) `V.s`: safety checks retained in the TCB; and (3) AEE implementation: runtime enforcement components. The evaluator can (1) open `scripts/TCB_stats.pdf`, and (2) review LOC breakdown for `V.a`, `V.s`, and AEE, and (3) optionally cross-reference the kernel source in `build/linux-6.7/`.

Interpretation: The verifier's approximation logic dominates its complexity. By shifting approximation out of the TCB and into smaller enforcements, AEE reduces the verifier's TCB by 4.5x, supporting Claim (C1).

(E2) Exploit Mitigation (30 min manual + 0.5 hr compute). This experiment evaluates AEE's ability to prevent exploits that bypass verifier checks. Exploit PoCs located in `exploits/` target known bugs in the verifier's approximation logic. To reproduce C2, the evaluator should (1) boot the baseline kernel using `scripts/boot.sh clean`, (2) SSH into the

VM and execute each exploit, (3) repeat this procedure on the AEE-enabled kernel. Please refer to `exploits/README.md` for more specific instructions.

Interpretation: On the unprotected kernel, the exploits compromise kernel integrity. Under AEE, violating accesses are caught at runtime, demonstrating effective mitigation. This validates Claim (C2), as shown in Table 2 and §6.1.

E3.1 Binary Size Overhead (1-2 hr compute). The goal is to quantify the increase in binary size of eBPF programs caused by AEE's runtime enforcement. The script `run_bench.sh bench_size` triggers execution of benchmarks inside QEMU for each kernel variant, measuring the bytecode size of eBPF programs before and after AEE transformations. It uses the disk image, runs the target workloads, and collects raw statistics. The `analyze.sh size` script parses the results and produces the analysis results.

```
./scripts/run_bench.sh bench_size
./scripts/analyze.sh size
```

Interpretation: Results show per-benchmark and average binary size increases. The "impacted average" reflects only modified programs, while the "overall average" includes all inputs. AEE causes a modest increase (~4.8%) consistent with the results shown in Table 3.

E3.2 Execution Time (1–2 hr compute). This part assesses AEE's runtime overhead during the execution of eBPF programs. The script `run_bench.sh bench_exec_time` launches the QEMU VM for each kernel variant, runs each benchmark suite, and saves execution statistics. The `analyze.sh exec_time` script computes per-suite and global averages.

```
./scripts/run_bench.sh bench_exec_time
./scripts/analyze.sh exec_time
```

Interpretation: The output includes execution times per benchmark and enforcement variant. The `all` configuration yields ~1.2% average runtime overhead, with individual enforcements contributing differently. These results support the statistics shown in Table 4.

E3.3 Verification Time (1–2 hr compute). Finally, we measure the increase in verification time incurred by enabling AEE. The `run_bench.sh bench_verify_time` script executes each benchmark and records the time spent on verification, including AEE transformations. The `analyze.sh verify_time` script computes total and average timing per kernel configuration.

```
./scripts/run_bench.sh bench_verify_time
./scripts/analyze.sh verify_time
```

Interpretation: The results show that while AEE increases verification time due to transformation overhead, the increase remains acceptable for practical deployment. Table 5 in the paper summarizes these findings.

Note: For additional guidance and expected outputs, please refer to `README.md` in the artifact archive.

A.5 Notes on Reusability

AEE is implemented as a modular set of patches applicable to Linux v6.7 and can be adapted to future versions with certain integration effort. Each enforcement mechanism—tag, offset, and size—is isolated, allowing evaluators to enable or disable them independently. The infrastructure is compatible with privileged execution contexts and exposes hooks (e.g., `BPF_F_TEST_REWRITES`) for custom instrumentation and introspection. Researchers may repurpose the artifact to develop or test alternative verifier-hardening techniques or runtime policies for eBPF safety enforcement.

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/usenixsec2025/`.