

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

OS-Sanitizer: System-wide Latent Defect Inference in Linux Applications

Addison Crump, Sahil Sihag, Florian Bauckholt, and Keno Hassler,
CISPA Helmholtz Center for Information Security; Thorsten Holz,
Max Planck Institute for Security and Privacy

https://www.usenix.org/conference/usenixsecurity26/presentation/crump

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: OS-SANITIZER: System-wide Latent Defect Inference in Linux Applications

Addison Crump
addison.crump@cispa.de
CISPA

Sahil Sihag
sahil.sihag@cispa.de
CISPA

Florian Bauckholt
florian.bauckholt@cispa.de
CISPA

Keno Hassler
keno.hassler@cispa.de
CISPA

Thorsten Holz
thorsten.holz@mpi-sp.org
MPI-SP

A Artifact Appendix

A.1 Abstract

Dynamic analysis is used throughout empirical testing, but primarily involves the detection of defects through the observation of corresponding faults; that is, something unwanted must happen for the defect to be detected. OS-SANITIZER is an eBPF-based tool for the passive detection of latent Linux application defects, observing events which indicate the capacity for a fault to manifest. Our prototype was evaluated quantitatively by measuring its performance in a benchmarking context and analyzing the report volume in a long-term deployment, and qualitatively by means of reimplementing existing strategies and actually using the tool to find defects in real software. Our provided artifact provides the prototype itself (including the reimplementation of existing work), experimental setup scripts, evaluation targets (i.e., the microbenchmarks and the scripts with which to measure performance on microbenchmarks, SPEC CPU 2017, and Browserbench), evaluation data of our own performance measurements, and the analysis scripts to filter and convert this data into LaTeX tables. We aim for all artifact evaluation badges based on our performance evaluation (Tables 2 and 3).

A.2 Description & Requirements

Our artifact represents a prototypical defect inference tool for Linux applications. As such, it must be executed within a Linux operating system environment. Our steps include the installing these tools within a virtual environment (e.g., within QEMU) running a specific Linux distribution as to avoid disruption of personal or organizational equipment.

A.2.1 Security, privacy, and ethical concerns

Our prototype requires no disabling of security mechanisms, does not interfere with the behavior of local programs, nor does it send any data to remote services. As such, there are no immediate privacy risks to users associated with the use of this tool.

At the same time, eBPF is not a mature software and is highly complex. During the preparation of this artifact, we identified multiple kernel versions where system integrity or availability could be affected due to bugs in the implementation of the userspace hooking of eBPF. Moreover, there is potential for sensitive data to be collected and stored *locally* within the report logs of our prototype. Finally, we cannot guarantee that no software supply chain attacks have occurred which expose the user to malicious code. To avoid potential system disruption and inadvertent collection of sensitive information, we suggest conducting all reproduction studies with our prototype within the proposed virtual environment. Further experimentation outside of this environment is strictly at the risk of the user.

A.2.2 How to access

Our stable artifact is made available under the Zenodo DOI:

https://doi.org/10.5281/zenodo.17979528

A "living" artifact is also made available on GitHub:

https://github.com/os-sanitizer/os-sanitizer

A.2.3 Hardware dependencies

The only requirement of evaluating our artifact is the use of an x86_64 host with at least 8 GB of RAM and the ability to launch x86_64 guests within a hypervisor. There may be slight differences on our evaluation hardware versus those used by the reviewers. Our reported figures are presented in terms of ratios, and we do not expect these values to fluctuate greatly.

A.2.4 Software dependencies

The most critical software dependency for the reproduction of our experiments is the SPEC CPU2017 benchmark. We

specify how this must be installed in Section A.3, but users must acquire a license for this software themselves.

Our software depends on many Fedora Linux packages, Rust programming language crates, and the Rust compiler. Provided that the virtual machine described in Section A.3 is correctly initialized with a network connection, all Rust dependencies will be automatically fetched during the build process. Furthermore, we provide a setup script which acquires all compilation dependencies within the artifact itself under `evaluation/setup_ae.sh`.

Users may utilize the hypervisor of their choice for conducting the experiments in a virtual environment, provided it is capable of running a typical Fedora Linux 42 installation.

A.2.5 Benchmarks

Beyond the SPEC CPU2017 benchmark described above, the microbenchmarks described in the paper may be found under the `examples/` directory in the artifact and the Browserbench "Speedometer 3.0" benchmark may be found at the URL: https://browserbench.org/Speedometer3.0/.

A.3 Set-up

As previously mentioned, users should first prepare a working hypervisor on their host system as a prerequisite. Then they may begin the installation steps below.

A.3.1 Installation

First, users should download the Fedora Linux 42 Workstation Edition ISO and the Fedora Linux 42 Server Edition ISO in their host environment. They should create two VMs, each with at least 4GB RAM, 30GB disk space, and a network connection, and on one install Fedora using the Workstation ISO using the typical steps and on the other the Server ISO.

With these installed, users may now download OS-SANITIZER on both machines with the following steps as root on each guest, replacing `$GIT_URL` with the GitHub URL provided above:

```
# dnf install git
# git clone $GIT_URL \
  --rev aad89c15c278e82200917314c0a045ea9fb7afe2
# evaluation/setup_ae.sh base
```

With these commands executed, the base installation is complete. Additional installation necessary for specific experiments is provided in the corresponding preparation step in Section A.4.2.

A.3.2 Basic Test

To ensure that OS-SANITIZER was installed correctly, reviewers can start the `os-sanitizer` utility with the following command:

```
# env RUST_LOG=info os-sanitizer --all
```

Within a few seconds, you should see alerts from local programs. The program will terminate within a few seconds of entering `CTRL-C`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): OS-SANITIZER induces low performance overhead during typical heavy workloads, though suffers when applied to frequently-invoked low-level userspace functions. This is demonstrated by experiments (E1) and (E2) and corresponds to Table 3 of the original paper.

(C2): OS-SANITIZER induces moderate performance overhead at the individual operation level and severe performance overhead for low-level userspace functions. This is demonstrated by experiment (E3) and corresponds to Table 2 of the original paper.

(C3): OS-SANITIZER induces negligible performance overhead during normal system usage. **We do not seek to reproduce this claim in this artifact evaluation** as this claim is qualitative and requires significant reviewer effort.

(C4): OS-SANITIZER identifies code regions likely to contain defects. **We do not seek to reproduce this claim in this artifact evaluation** as this claim is qualitative and requires significant reviewer effort. Reviewers who seek to replicate these findings may do so by installing the corresponding vulnerable versions of the softwares mentioned within the appropriate VM, e.g., by installing Docker version 26.1.3, or by viewing reports emitted by microbenchmarks provided under `examples`.

A.4.2 Experiments

(E1): [SPEC CPU2017] [30 human-minutes + 5400 compute-minutes for `-all` + 10GB disk] Establishes the performance impact of OS-SANITIZER during industry-standard heavy workloads that one might observe on servers and developer machines. This experiment takes place on the Server VM.
Preparation: Ensure that you have followed all installation instructions for the Server VM. Then, place the SPEC CPU2017 installer ISO file on the machine, then run

```
# evaluation/setup_ae.sh spec <spec.iso>
```

This script has been tested with `cpu2017-1.1.0.iso` (MD5: `f26e4f9404c31b82d40be2d71971d1c0`).
Execution: The scripts inside `evaluation/spec/` will

launch OS-SANITIZER with the correct configuration for that experiment. In total we have 18 different evaluation scripts which we recommend running in a VM of its own. For example, `eval_SPEC_intspeed_00.sh` should be copied to base directory where SPEC CPU2017 is installed and launched with `bash -x eval_SPEC_intspeed_00.sh`. Consequently, this will populate `cpu_intspeed_log_0_all_...` in base SPEC directory as well as results in child `result` directory.

Results: Files from 18 different VMs need to be arranged in the directory structure described in `evaluation/filtering/filter_SPEC.py`. Execution of `filter_SPEC.py` will populate filtered results into `evaluation/data/data_SPEC.json`. Additionally, Browserbench experiments needs to be performed to regenerate Table 3 from the paper as described in **E2**.

Due to time constraints, we recommend executing the "no passes (17)", "reference policy (16)", "all passes (00)", and two additional configurations of the reviewer's choice on three benchmarks of the reviewer's choice, instead of the ten benchmarks of the original paper. This reduces reviewer workload without excessively sacrificing rigor, as we have no mechanism by which to predict which experiments will be selected.

(E2): [Browserbench] [10 human-minutes + 3 compute-hours + 5GB disk] Establishes the performance impact of OS-SANITIZER on heavy browers-based workloads that one might observe on user-facing devices. This experiment takes place on the Workstation VM as it requires a *headful* environment.
Preparation: Ensure that you have followed all installation instructions for the Workstation VM. To install Browserbench specific dependencies, then execute:

```
# evaluation/setup_ae.sh browserbench
```

Execution: This experiment may be launched directly with the following commands.

```
# cd evaluation/browserbench
# bash -x eval_puppet_speedometer3.sh
# python3 ../filtering/filter_browserbench.py
    results/
```

Results: The `filter_browserbench.py` file aggregates results to the `evaluation/data/data_speedometer.json`, which is used in the shared output script (`output_SPEC_and_browser_print.py`). Since the paper uses a combined table for data from E1 and E2, both files in `evaluation/data` have to present.

(E3): [Microbenchmarks] [30 human-minutes + 1 compute-hour + 1GB disk] Establishes the performance impact of OS-SANITIZER on individual operations by means of benchmarking short sequences of operations repeatedly.
Preparation: Ensure that you have followed all installation instructions for the Server VM.
Execution: This experiment may be launched directly with the following commands.

```
# cd evaluation/microbenchmark
# bash eval_microbench.sh | \
    tee stdout_of_eval_microbench
# bash ../filtering/filter_microbench_data.sh
```

Results: Run the Python script `../output/output_microbenchmark_print.py` to generate the LaTeX code for Table 2 in the paper.

A.5 Notes on Reusability

OS-SANITIZER is merely a prototype to demonstrate the capacity of eBPF to be used to identify likely defective code regions. Alas, it is just that: a prototype. There are several usability gaps which we would like to address in the future.

For now, this prototype is best used via systemd units. A description of how to install OS-SANITIZER on a system for purposes other than evaluation, including installation as a systemd unit, is provided in the root README of the artifact's repository. Users intending to build their own passes are highly suggested to first review simpler passes like `system_absolute` pass under `os-sanitizer-ebpf`, observing how the information flows from eBPF to userspace via the components provided in the `os-sanitizer-common` crate, and finally is processed in userspace in the main loop of the `os-sanitizer` crate. More advanced passes will likely require the use of advanced helper functions, for which refer users to the corresponding Aya documentation and the examples provided by Michal R (AKA vadorovsky). We are happy to provide further support via the Discussions tab of the artifact repository, and will happily accept further contributions that make the development and usage of OS-SANITIZER easier in the future.

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/usenixsec2026/.