

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Concurrency Fuzzing of the Linux Kernel with eBPF

Jiacheng Xu, *Zhejiang University;* Dylan Wolff, Xing Yi Han, Jialin Li, and Abhik Roychoudhury, *National University of Singapore*

https://www.usenix.org/conference/usenixsecurity26/presentation/xu-jiacheng

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: Concurrency Fuzzing of the Linux Kernel with eBPF

Jiacheng Xu*
Zhejiang University†

Dylan Wolff*‡
National University of Singapore

Xing Yi Han
National University of Singapore

Jialin Li
National University of Singapore

Abhik Roychoudhury
National University of Singapore

Abstract

Concurrency is indispensable for modern software systems to meet performance and scalability demands, yet concurrency bugs remain notoriously difficult to detect and reproduce. Controlled Concurrency Testing (CCT) mitigates this challenge by systematically exploring thread interleavings through scheduling control. However, existing CCT approaches for OS kernels largely rely on external enforcement mechanisms, such as custom hypervisors or invasive kernel patches, resulting in substantial overhead and limited maintainability and extensibility. In this work, we present SECT, the first kernel-native concurrency fuzzing framework that rethinks scheduling as a first-class exploration mechanism. SECT introduces a novel CCT scheduler with temporal isolation scheduling and embeds programmable scheduling policies directly into the kernel dispatch path via eBPF, enabling fine-grained control over thread interleavings without customized hypervisors or extensive kernel core modification. In addition, SECT provides a preemption-safe instrumentation mechanism for injecting scheduling points at critical kernel events and incorporates a two-phase fuzzing workflow to jointly explore both sequential and concurrent behaviors. Our evaluation demonstrates that SECT achieves 38% more branches, 57% overhead reduction and 11.4× speed-up in bug exposure compared to a leading state-of-the-art kernel concurrency fuzzer. Moreover, SECT discovers eight previously unknown concurrency-related bugs in the Linux kernel, six of which have already been confirmed and fixed by developers.

A Artifact Appendix

A.1 Abstract

This artifact contains SECT, as well as infrastructure for setting it up on a recent target kernel (v6.13-rc4), our benchmark of kernel concurrency bugs used in our evaluation, and data from our real-world bug-finding runs.

A.2 Description & Requirements

Generally, the system requirements for this artifact are (1) a relatively recent version of Ubuntu (tested on 24.04) (2) Docker (3) QEMU with KVM enabled. In terms of hardware recommend a machine with at least 8 logical CPU cores and 16GB of RAM. All other setup should be handled by scripts in the SECT repository.

A.2.1 Security, privacy, and ethical concerns

The risks of using SECT are minimal, especially if used within a VM, as is the documented setup in our artifact. If you attempt to run SECT on your host system outside of a VM, any resulting kernel panics will of course affect your host system directly.

A.2.2 How to access

Our artifact is available open-source on Github at `https://github.com/m0ck1ng/sect` and also permanently archived at `https://doi.org/10.6084/m9.figshare.32468280`

A.2.3 Hardware dependencies

SECT does not require any specific hardware, though we recommend a multi-core system with at least 8 logical cores for performance when running fuzzing campaigns and to ensure that the Linux kernel itself can be rebuilt by your system in a reasonable amount of time. To reproduce the full fuzzing experiments, we recommend a machine with at least 80 cores and 2GB of RAM per core, to allow all 10 trials to be run in parallel.

*Both authors contributed equally
†Research conducted primarily at the National University of Singapore
‡Corresponding Author: `wolffdy0@gmail.com`

A.2.4 Software dependencies

We have developed and tested SECT on Ubuntu 24.04. Additional recent versions of Ubuntu or Debian are also likely to work without modification. Other Linux distributions may also work, but are untested and will require some slight modifications of shell scripts (e.g. to install dependencies from `rpm` instead of `apt`).

Additionally, the host system needs Docker installed to use our provided setup. Docker is used purely for a hermetic build environment to ensure all other dependencies are the correct versions. For example, SECT and the kernel under test are mounted into a docker container as a volume, and built inside the container rather than on the host system.

The fuzzing setup uses a forked version of syzkaller, which runs in one or more virtual machines using QEMU. QEMU is installed as part of the build script automatically on the host system via `apt`.

Finally, to create a disk image for the guest VM, `debootstrap` is also a required dependency, installed automatically by the main build script.

A.2.5 Benchmarks

We have provided the details of the ten concurrency bugs in the benchmark from our experiments in our artifact, including reproducing inputs and a kernel patch for v6.13-rc4 which introduces all of the bugs into that kernel version.

A.3 Set-up

All that is needed for setup is to clone the SECT repository:

```
git clone https://github.com/m0ck1ng/sect.git
```

And to install Docker:

https://docs.docker.com/engine/install/

A.3.1 Installation

We have consolidated the installation into a single script:

```
./build.sh
```

Note that some operations may require superuser permissions to e.g. install dependencies or mount a volume for disk image creation.

A.3.2 Basic Test

The best way to check that SECT is working after installation is to start a fuzzing campaign via:

```
./syzkaller/bin/syz-manager
-config configs/syzkaller.cfg.example
```

The configuration should have been pre-populated by the installation process, but it can be edited according to your system requirements as necessary. You should see output at the command line and in the web interface (`http://0.0.0.0:56741`) containing:

```
Using thread scheduler: <path>
```

and later you should see that `schedCollided` increases over time, e.g.:

```
VMs 4, executed 2434, schedCollided 650, cover
29526, signal 38453/47876, crashes 0, repro 0,
triageQLen 8817
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SECT achieves higher fuzzing throughput and coverage that other tools, shown in Figure 3 and Table 2.

(C2): SECT is more effective at finding concurrency bugs than the native scheduler under Syzkaller or SegFuzz, shown by Table 3 and Figure 4.

A.4.2 Experiments

(E1) This experiment reproduces the code coverage and throughput results for SECT (C1) as well as the Bug-Finding Results (C2) shown in Table 3. Please note that for all experiments to finish in 48 hours, all trials must be run simultaneously, requiring at least 80 CPU cores for SECT (4 VM instances * 2 cores per instance * 10 trials).

[6 Hours Human] [48 Hours Compute]

1. *Preparation.*Build the benchmark kernel with all 10 concurrency bugs by applying the patch provided in our artifact and building SECT with the provided script.
2. *Experiment.* Run a fuzzing campaign via `syz-manager` as outlined in the artifact using the provided configuration for each *trial*, using a timeout of 48 hours.
3. *Results.* For each crash found by SECT, manually check if it corresponds to any of the bugs in the benchmark. Note that once a bug has been correlated, any crashes from other trials with the same hash can also be mapped to that bug. We compute throughput and latency from the total number of executions reported by `syz-manager` and the 48-hour time period. To obtain coverage data we use the documented utility provided by Syzkaller: https://github.com/google/syzkaller/blob/master/docs/coverage.md

We note that for comparing with Syzkaller, most of the above steps are identical, save for using a clean build of Syzkaller in lieu of SECT. For SegFuzz, we use the instructions provided by the tool authors. For Snowboard, we similarly use the instructions provided by the tool authors.

(E2) This experiment reproduces the survival plots in Figure 4 (C2). Unfortunately, we didn't find a good way to automate these experiments, so they are currently interactive and thus take a very long time to reproduce.
[>10 Hours Human + Compute (interactive)]

1. *Preparation.* Build the benchmark kernel with all 10 concurrency bugs by applying the patch provided in our artifact and building SECT with the provided script.

2. *Experiment.* Start a VM instance using the script provided in our artifact `./run_qemu.sh`. Copy all necessary binaries and programs into the guest with the provided script `./copy.sh`. Use SSH to obtain a second terminal window inside the guest. In one window, start the sched-ext scheduler `./scx_serialise` with the desired scheduling algorithm via `-r`. In the other window, execute the desired reproducer with `syz-execprog` using two processes and 10000 repetitions.

3. *Results.* For each crash, we record the number of schedules until the crash as logged by the window with `scx_serialise`. We generate the survival plots using the script provided in our artifact (`scripts/analysis/plot.py`).

A.5 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/.