USENIX Security '26 Artifact Appendix: KernelRCA: Facilitating Root Cause Analysis of Memory Corruptions in Linux Kernel with Contextual Causality Chain

Kangzheng Gu
Fudan University

Yifan Zhang
Fudan University

Yuan Zhang
Fudan University

Min Yang
Fudan University

A Artifact Appendix

A.1 Abstract

This artifact contains the complete implementation of KernelRCA, an automatic root cause analysis system for Linux kernel memory corruptions. It consists of three components: (1) the system implementation of KernelRCA; (2) the dataset used in the paper's evaluation; and (3) evaluation scripts for measuring the performance of KernelRCA. These components enable the reproduction of the main results reported in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is available on Zenodo at https://zenodo.org/records/18410035. Please use the latest version of this record. Alternatively, the implementation is also available on GitHub at https://github.com/seclab-fudan/KernelRCA and https://github.com/xiaoguai0992/KernelRCA. Note that the GitHub repositories include only the implementation of KernelRCA and the dataset, and do not contain the evaluation scripts used to reproduce the main results reported in the paper.

A.2.3 Hardware dependencies

To run KernelRCA, the underlying CPU must be an Intel platform with KVM available. The artifact requires a machine with a minimum of 16 CPU cores, 32 GB of RAM, and 500 GB of disk storage. Increasing the number of CPU cores can accelerate Linux kernel compilation, thereby reducing the end-to-end execution time of KernelRCA.

A.2.4 Software dependencies

KernelRCA is evaluated on Ubuntu 18.04. We recommend installing the system using the image available at https://releases.ubuntu.com/18.04/ubuntu-18.04.6-desktop-amd64.iso (kernel version 5.4.0-84-generic). Other Ubuntu versions may cause failures in S2E tracing. The `README.md` included in the artifact provides detailed instructions for installing all required software dependencies. An active Internet connection is required during the installation and execution of KernelRCA.

KernelRCA **cannot** be deployed in a Docker environment. If KernelRCA is installed inside a virtual machine, please ensure that KVM nested virtualization is correctly enabled on the host system so that the virtual machine can access hardware-assisted KVM acceleration.

A.2.5 Benchmarks

The dataset is included in the artifact.

A.3 Set-up

A.3.1 Installation

Before installation, we recommend using a clean installation of Ubuntu 18.04. Detailed installation instructions of KernelRCA are provided in the `README.md` included in the artifact.

A.3.2 Basic Test

After installation, navigate to the `KernelRCA/rca` directory and execute the following command:

```
python3 run.py crash_b66d8de2cec1e3878a0524807b93d96bba182fba
```

This command performs root cause analysis on the example vulnerability. Upon completion, the output report is expected to be generated at

```
KernelRCA/s2e/projects/
    crash_b66d8de2cec1e3878a0524807b93d96bba182fba/
    report.txt
```

The file `report.txt` contains the final root cause analysis report produced by KernelRCA.

If an error occurs during the image build process in KernelRCA, a helpful debugging approach is to examine the build logs and identify the first error that appears. A common cause is network-related issues during the Docker image build or the Debian image build either inside or outside the VM. Problems occurring inside the VM can be diagnosed by connecting to the VM through the VNC interface. If users wish to rerun the image build from scratch, they should remove all files under `s2e/images/*`, particularly the `.stamps` and `.tmp-output` directories within this path.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): KernelRCA successfully generates root-cause reports for the 54 vulnerabilities described in Section 4.2.

(C2): KernelRCA outperforms existing techniques, i.e., crash reports and Syzbot Cause Bisection, in terms of the root-cause distance metric defined in Section 4.2. The comparative results are presented in Table 3.

(C3): The overhead statistics of KernelRCA, described in Section 4.3.1 and illustrated in Table 8, show that the average trace size is 1.39 GB and diagnosing a bug takes approximately 97 seconds on average.

A.4.2 Experiments

(E1): *[Effectiveness] [2 human-hour + 24 compute-hour (all cases, about 30 minutes per-case) + 500 GB disk]:*
How to: Install the KernelRCA, perform the root cause analysis for vulnerabilities in the dataset, and review the root-cause report. Since performing root cause analysis on all cases is time-consuming, we recommend sampling a subset of cases to run and using their results to calculate statistics for the remaining evaluations.
Preparation: Build and install KernelRCA following the instructions in `README.md`. Ensure that the basic test produces the expected root cause report `report.txt`.
Execution: (Option 1, recommended) Navigate to the `KernelRCA/rca` directory and execute:

```
python3 run.py <crash_xxxx>
```

This command will automatically analyze the specified vulnerability and generate a root cause report.
(Option 2) Navigate to the `KernelRCA/rca` directory and execute:

```
python3 run_batch.py
```

It will automatically run all the available cases. Note that some cases, such as `4b1e841`, may generate very large traces (hundreds of gigabytes), which can consume significant time and storage, potentially causing the analysis to appear stalled. To skip such cases, you can manually create a folder named `KernelRCA/s2e/projects/<crash_id>` and place an empty `report.txt` file inside it. The system will then detect and skip the designated case.
Results: The results will be saved in `KernelRCA/s2e/projects/<crash_id>/report.txt`. Users need manually review the root cause report and assess its correctness according to Section 4.1 of the paper.

(E2): *[Comparison with Baselines] [5 human-minute + 10 compute-minute (per-case)]:*
How to: Run the scripts to calculate and compare the root-cause distances of KernelRCA and baselines for a given case.
Preparation: Ensure that KernelRCA has been run on the designated case and that the `report.txt` file is available.
Execution: Navigate to the `KernelRCA/eval` directory and run:

```
./eval_distance.sh <crash_id>
```

This command automatically calculates the root-cause distances for KernelRCA's result and the baselines'. The output is saved in `KernelRCA/s2e/projects/<crash_id>/distance` and will look like:

```
kernelrca_distance: 12
syzbot_distance: 123456
crashreport_distance: 12345
```

Note that the reported distances may vary from those in the paper, as execution traces can differ between runs. In most cases, KernelRCA's root-cause distance should be smaller than those of the baselines, supporting claim (C2).

(E3): *[Efficiency] [1 human-minute + 1 compute-minute (per-case)]:*
How to: Run the scripts to calculate the overhead of KernelRCA for a designated case.
Preparation: Ensure that KernelRCA has been executed on the specified case and that the `report.txt` file is available.
Execution: Navigate to the `KernelRCA/eval` directory and run:

```
python3 eval_overhead.py <crash_id>
```

This command automatically calculates the overhead of KernelRCA for the specified case. The output will be similar to:

```
Trace Size: 6.14 (GB)
Total Time: 299 (s)
Tracing Time: 171 (s)
Norm Time: 36 (s)
Context Rebuild Time: 67 (s)
RCA Time: 25 (s)
```

Note that these statistics may vary from those reported in the paper due to differences in execution traces between runs and variations in hardware environments. On average, the results should support claim (C3) when the conditions described in the paper are met (which do not correspond to the minimal environment requirements). If you find that the execution time for a case is significantly longer than reported (e.g., around 30 minutes or more), this is likely because most of the time is spent on kernel compilation. Alternatively, user can run `run.py` with parameters such as `--only rca` to execute only a single analysis step. This execution time should be closer to the numbers reported in the paper.

A.5 Notes on Reusability

The functionality of KernelRCA is modularized to promote reusability. For example, by passing the argument `-only trace` to `run.py`, KernelRCA performs tracing only. The dynamically collected tracing data are organized in Protobuf format, which may be useful for other program analysis tasks such as taint analysis, dynamic program slicing, and more.

To adapt KernelRCA to kernel versions outside the 5.0–5.14 range, users need to extend `rca/patch_kernel.py` to patch the kernel accordingly, enabling S2E instrumentation as well as KernelRCA's instrumentation to capture the required runtime data.

If users wish to apply KernelRCA to additional types of vulnerabilities by adding new detection rules for unexpected program behaviors, they can refer to `rca/analyzer/cpp/src/RootCauseAnalyzer.cc` to implement their own detectors.

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/.