Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples

Dandan Xu, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China;* Di Tang, Yi Chen, and XiaoFeng Wang, *Indiana University Bloomington;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China;* Haixu Tang, *Indiana University Bloomington;* Longxing Li, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China*

https://www.usenix.org/conference/usenixsecurity24/presentation/xu-dandan

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

USENIX Security '24 Artifact Appendix: Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples

Dandan Xu[1,2], Di Tang[3], Yi Chen[3], XiaoFeng Wang[3], Kai Chen[1,2], Haixu Tang[3], Longxing Li[1,2]
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Indiana University Bloomington
{xudandan, chenkai, lilongxing}@iie.ac.cn, {tangd, chen481, xw7, hatang}@iu.edu

A Artifact Appendix

A.1 Abstract

RACING is an efficient statistical Root-Cause Analysis (RCA) solution that employs reinforcement learning on *counterexamples* for acceleration. In this artifact, we provide RACING's source code and automation scripts for analyzing the 30 testing programs used in our evaluations. The code and programs have been tested to work on a 64-bit Ubuntu 20.04 server with 32 Intel Xeon(R) Silver 4110@2.10 GHz CPU cores and 128 GB memory. Please note that due to the nature of fuzzing, a machine with more powerful CPUs can potentially achieve better results than those reported in the original paper.

This appendix contains the necessary steps for setting up a proper testing environment, along with a detailed manual for reproducing the major results in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The code is available via `https://github.com/0xdd96/Racing-code/releases/tag/artifact-evaluation`.

A.2.3 Hardware dependencies

The results in our paper were obtained on a server with 32 Intel Xeon(R) Silver 4110@2.10 GHz CPU cores and 128 GB memory. Note that our current implementation of RACING is *single-threaded*. Therefore, the analysis time for a *single vulnerability* is primarily determined by the CPU's frequency. Nevertheless, for the 30 vulnerabilities in our testing dataset, one can start multiple instances of RACING on multiple cores to reduce the total running time of the experiments.

A.2.4 Software dependencies

RACING was evaluated on a 64-bit Ubuntu 20.04 server. To ensure a consistent testing environment, we provide a `Dockerfile` that initializes a fresh Ubuntu 20.04 OS container and installs the necessary software dependencies. Please ensure your machine supports `Docker` and can run the `Ubuntu:20.04` image. Additionally, ensure you have *root* access to your host machine as required by the setup of RACING.

A.2.5 Benchmarks

Our evaluation dataset contains 30 vulnerabilities. These vulnerabilities were found in 21 programs with varying scales, ranging from 857 to 980,019 lines of source code. The `examples` folder of our GitHub Repository contains the necessary data for reproducing the 30 vulnerabilities, with each vulnerability stored in a separate folder numbered following the IDs in Table 2 & Table 3 of our paper. For each vulnerability, we provide a PoC that triggers it in the `seed` folder, along with 4 automated scripts to reproduce the results:

1. `01_build_trace.sh`: downloads the source code of the vulnerable program (via Internet), and builds a debug version of the program for tracing.
2. `02_PocExecutionInspector.sh`: executes the vulnerable program using the PoC as input, using Intel PIN to collect execution traces, then maps binary addresses to source code locations using two Python3 scripts `tracing.py, addr2line.py`.
3. `03_build_fuzz.sh`: builds an instrumented program for RACING's fuzzing process.
4. `04_racing.sh`: starts RACING's fuzzing process, and produces a ranked list of predicates in the end.

A.3 Set-up

Before you start everything, make sure to set the following configurations on your host machine (for AFL & statistical analysis):

```
# use root permission if necessary
$ echo core >/proc/sys/kernel/core_pattern
$ cd /sys/devices/system/cpu
$ echo performance | tee
 ↪  cpu*/cpufreq/scaling_governor
# disable ASLR
$ echo 0 | tee
 ↪  /proc/sys/kernel/randomize_va_space
```

A.3.1 Installation

1. Clone RACING's GitHub Repository:

   ```
   $ git clone
    ↪  https://github.com/0xdd96/Racing-code
   ```

2. RACING was evaluated on Ubuntu 20.04. Our repository provides a Dockerfile that handles the installation of necessary dependencies and the compilation of RACING.

   ```
   $ docker build -t racing-eval:latest .
   ```

3. Start RACING's container. Note that all subsequent steps should be carried out within the `racing-eval` container.

   ```
   docker run --name racing-eval --init -d -v
    ↪  $PWD/examples:/Racing-eval/examples
    ↪  racing-eval:latest tail -f /dev/null
   ```

A.3.2 Basic Test

Here we use V_{20} (CVE-2020-19497 in `matio:bcf0447`) from our dataset (`/Racing-eval/examples`) to showcase the basic usage of RACING.

1. *[15 compute-seconds]* Compile a `matio` binary with debugging symbols for tracing:

   ```
   $ docker exec -ti racing-eval bash
   $ cd ./examples/20-matio-integer-overflow
   $ ./01_build_trace.sh
   ```

2. *[5 compute-seconds]* Inspect PoC execution:

   ```
   $ ./02_PocExecutionInspector.sh
   ```

3. *[40 compute-seconds]* Compile an instrumented `matio` binary for fuzzing:

   ```
   $ ./03_build_fuzz.sh
   ```

4. *[15 compute-seconds]* Start RACING's fuzzer and wait for it to terminate. The ranking results are stored in the `ranked_file` in the `afl-workdir-batch0` directory:

   ```
   $ ./04_racing.sh
   ```

A.4 Evaluation workflow

RACING was evaluated using 30 vulnerabilities. we provide two scripts `run-all.sh` and `check_all.py` to automate the running of the experiments and the analysis of the final results.

A.4.1 Preprocessing

[1 human-minute + 30∼60 compute-minutes + 8.5 GB disk] Before evaluating RACING on the 30 test cases, you should invoke the following script to download source code and build instrumented binaries. Our script provides a `-j` flag that spawns multiple threads to speedup this process.

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ ./run-all.sh -j${NUM_THREADS} build
```

Note that the script may fail to download source code due to network errors. You can check `01_build_trace.log` for details and re-run the build script to compile the remaining test cases.

A.4.2 Major Claims

(C1): RACING *significantly improves the efficiency of statistical RCA. This is proven by the experiments (E1) in Section 5.2, whose results are illustrated in Table 3.*

(C2): RACING *achieves high rankings for root cause analysis. This is proven by the experiment (E2) in Section 5.2, whose results are illustrated in Table 3.*

A.4.3 Experiments

(E1): *[2 human-minutes + 12 compute-hours + 30GB disk]: This experiment will use* RACING *to analyze the root causes of 30 vulnerabilities, measuring the running time of each test case.*

How to: *First invoke the* `run-all.sh` *script to start analyzing the 30 vulnerabilities (the multi-threading flag* `-j` *is optional). The script should output the running time of each test case when it's finished. After all the test cases have been analyzed, please check* `overall_status.run.log` *for the running time of* RACING.

Preparation: *The preprocessing step in appendix A.4.1 is obligatory to obtain correct results.*

Execution: *Execute the following commands in the* `racing-eval` *container.*

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ ./run-all.sh -j${NUM_THREADS} run
$ cat overall_status.run.log
```

Results: *The output of* `overall_status.run.log` *should match* RACING*'s execution time (T_{all}) in Table 3 (with acceptable deviations) in Section 5.2 of the paper.*

(E2): *[1 human-minute + 2 compute-minutes]: This experiment will use* `check_all.py` *to analyze the final rankings of the root causes for the 30 vulnerabilities.*

How to: *Please run* `check_all.py` *to collect the final rankings of the root causes. The ground truths are stored in* `examples/ground_truth.txt`.

Preparation: *The preprocessing step in appendix A.4.1 and the steps in* **E1** *are obligatory to obtain correct results.*

Execution: *Execute the following commands in the* `racing-eval` *container.*

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ python3 check_all.py check_all
 ↪  ground_truth.txt $PWD
```

Results: *The output of* `check_all.py` *should match* RACING*'s rankings in Table 3 (with acceptable deviations) in Section 5.2 of the paper.*

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/usenixsec2024/`.