USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

RangeSanitizer: Detecting Memory Errors with Efficient Range Checks

Floris Gorter and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

https://www.usenix.org/conference/usenixsecurity25/presentation/gorter

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: RangeSanitizer: Detecting Memory Errors with Efficient Range Checks

Floris Gorter
Vrije Universiteit Amsterdam
f.c.gorter@vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

A Artifact Appendix

A.1 Abstract

In this artifact we provide instructions to reproduce our main results. This artifact concerns our tool named RangeSanitizer (RSan), which detects spatial and temporal memory errors in C/C++ programs. Our results show that RSan can successfully detect memory errors, and does so with higher runtime performance than traditional redzone-based bug sanitizers. We have validated the artifact using a system containing an Intel i9-13900K CPU and running Ubuntu 22.04 with a stock Linux v5.15 kernel. Our source code is available at: https://github.com/vusec/rangesanitizer

A.2 Description & Requirements

The experimental setup of RSan (with implicit pointer tagging by default) concerns an x86 system running Ubuntu. Using explicit pointer tagging requires hardware support. We require the evaluators to obtain the SPEC CPU benchmarking suite themselves, since we cannot distribute the licensed software.

A.2.1 Security, privacy, and ethical concerns

As a bug sanitizer, RSan poses no risks to the security and privacy of the target machine, and has no ethical implications.

A.2.2 How to access

The files for the artifact evaluation are currently available at: https://github.com/vusec/rangesanitizer/releases/tag/ae and https://zenodo.org/records/14701524

A.2.3 Hardware dependencies

By default, we evaluate the implicit pointer tagging design that supports any (legacy) x86 architecture. If desired, RSan's explicit pointer tagging can be used, which requires CPU hardware support: either Arm TBI (Armv8-a and onwards) or Intel LAM (Lunar/Arrow Lake). In order to support some of the benchmarks, 32 GB of RAM is recommended.

A.2.4 Software dependencies

While RSan does not have very specific software requirements, some packages from the Ubuntu package manager are required to be installed to build RSan's dependencies (e.g., the LLVM project). These are described in the Set-up section. We evaluated RSan on Ubuntu 22.04.

A.2.5 Benchmarks

For this artifact we benchmark using the SPEC CPU2006 benchmarking suite, and the Juliet test suite.

A.3 Set-up

We recommend using a bare-metal desktop system with (at least) 32 GB of RAM, running Ubuntu 22.04, glibc 2.35, and a stock v5.15 Linux kernel.

A.3.1 Installation

1. Obtain the artifact source from the `ae` release:

```
git clone \
https://github.com/vusec/rangesanitizer.git \
 --recurse-submodules --branch ae
cd rangesanitizer
```

2. Install some standard dependencies:

```
sudo apt install ninja-build cmake gcc-9 \
autoconf2.69 bison build-essential flex \
texinfo libtool zlib1g-dev unzip

pip3 install psutil terminaltables
```

3. Configure the RSan environment by editing the `env.sh` file and modifying the `RSAN_TOP` variable to reflect the working directory of the system, and then run:

```
source env.sh
```

4. We recommend changing the CPU scaling governor to `performance` for stability in benchmarks (requires sudo):

```
echo "performance" | sudo tee \
 /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

5. Install the RSan infrastructure by running:

```
./install-all.sh
```

NOTE: installing LLVM can take up a lot of RAM when using multiple cores. If the compilation process crashes (OOM), modify the `ninja -j <cores>` parameter inside `install-all.sh` to use fewer cores.

A.3.2 Basic Test

To test the basic functionality of RSan, we provide two test programs in the `examples` directory. To compile and run these tests, execute:

```
cd examples
./test-implicit.sh
```

See README.md in the artifact for the exact expected output format (returned addresses may vary).

A.4 Evaluation workflow

In order to run the Juliet test suite and SPEC CPU along with its benchmarks we make use of a public infrastructure under the `infra` directory. The infra also makes sure the SPEC binaries are pinned to core 0. Make sure that the necessary python packages are installed (see the Installation section).

A.4.1 Major Claims

(C1): RSan can detect spatial and temporal memory errors bounded by its security guarantees (as described in Section 7.1). This is proven by experiment E1.

(C2): RSan provides high performance in terms of runtime overhead (see Section 7.3). This is proven by experiment E2.

A.4.2 Experiments

(E1): [30 compute-minutes]: Confirming memory error detection.

How to: The Juliet test suite contains buggy programs for which RSan can detect the bugs at runtime.

Preparation: Ensure that the `env.sh` and `install.sh` scripts have been executed successfully. Note that we run Juliet with optimization flag -O0, because standard optimizations (e.g., -O2) hide bugs in the test cases. Feel free to also test Juliet with AddressSanitizer (ASan): change the target of setup.py from `rsan-impl_O0` to `asan_O0` in the command below.

Execution: Execute the following command, which builds and runs the relevant Juliet categories (see Table 1):

```
python3 setup.py run juliet rsan-impl_O0 \
--build --parallel=proc \
--parallelmax=$(nproc) \
--cwe 121 122 124 126 127 415 416
```

Results: The exact expected output can be found in the README.md file in the artifact. All tests are expected to pass.

(E2): [5 compute-hours]: Confirming runtime performance.

How to: Run the SPEC CPU2006 benchmarking suite instrumented by RSan and ASan, and observe the performance overhead compares to a baseline (non-instrumented) run.

Preparation: SPEC CPU2006 needs to be available on the system and the `RSAN_SPEC2006` variable in `env.sh` needs to point to the directory where it is installed. For the artifact evaluators, if they cannot obtain SPEC CPU2006, we can provide access to a machine ready to run SPEC. Ensure that the `env.sh` and `install.sh` scripts have been executed successfully (`env.sh` needs to be reloaded after modifying its content).

Execution: Execute the following command, which builds and executes SPEC CPU2006 for three runs: the baseline, one with ASan, and one with RSan, which in total takes multiple hours:

```
python3 setup.py run spec2006 baseline_O2 \
rsan-impl_O2 asan_O2 --build \
 --parallel=proc --parallelmax=1
```

Results: The exact expected output can be found in the README.md file in the artifact. Note that there will be a corresponding output folder in the `results` directory which can be inspected. To obtain a summary of the results from the SPEC CPU2006 runs, again make use of the `setup.py` script. Execute the following command:

```
python3 setup.py report spec2006 \
results/last --field runtime:median \
maxrss:median
```

The output of this command can then be used to calculate the runtime and memory overheads for each individual binary, as well as for the geomean. As reported in Figure 9: if ran on an x86 machine with an i9-13900K CPU, the expected runtime overhead for RSan is 51%, and 95% for ASan. See the README.md file in the artifact for a complete example output and the corresponding geomean overhead calculation.

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/usenixsec2025/.