USENIX Security '26 Artifact Appendix: CombiSan: Unifying Software Sanitizers for Comprehensive Fuzzing

Matteo Marini*
Sapienza University of Rome
m.marini@diag.uniroma1.it

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

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

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

* Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

This artifact provides the instructions to run and evaluate CombiSan, a fuzzing-optimized software sanitizer that combines detection of addressability issues, use-of-uninitialized-memory (UUM) errors, and other undefined behavior bugs. Our results show that CombiSan can do so with relatively low overhead, while matching the accuracy of the three most used software sanitizers (ASan, MSan, and UBSan).

A.2 Description & Requirements

The artifact consists of a modified LLVM 20.1.0 framework, which is used to instrument the targets to enable bug detection, a modified AFL++ 4.32c project that supports *deferred detection* of bugs, and infrastructure to test CombiSan. The performance tests require the SPEC CPU benchmarking suite, which is a licensed product we cannot redistribute.

A.2.1 Security, privacy, and ethical concerns

This artifact does not pose concerns to evaluators. Broader ethical considerations are discussed in the paper.

A.2.2 How to access

The source code of CombiSan, and the infrastructure to test it, is available at https://github.com/vusec/combisan, where it will be maintained. The GitHub `README.md` and `AE.md` pages contain more instructions. We provide permanent storage at https://zenodo.org/records/16949365.

A.2.3 Hardware dependencies

CombiSan supports commodity hardware. We recommend at least 16 GB of RAM to build LLVM. Additionally, we recommend at least 50 GB of disk space for the installations of LLVM and AFL++, as well as the Docker images.

A.2.4 Software dependencies

CombiSan can run on any modern installation of a Linux-based OS that is capable of compiling and running LLVM 20.1.0 and AFL++ 4.32c, such as any recent Ubuntu version ($\geq$20.04). We evaluated on Ubuntu 24.04 LTS. To support the compilation of its components, common development tools are required (e.g., `gcc`/`g++`, `cmake`, `ninja`); they can be easily found in every modern package manager. For example, using

```
sudo apt install gcc g++ cmake ninja-build \
unzip python3-terminaltables
```

To facilitate fuzzing, we used Docker; this artifact contains the Dockerfiles and accompanying build instructions.

A.2.5 Benchmarks

CombiSan was evaluated using three major benchmarks. First, performance was evaluated using the SPEC CPU benchmarks. Then, we assessed accuracy capabilities using the NIST Juliet Test Suite. Lastly, we tested fuzzing capabilities using a custom dataset of common fuzzing targets.

A.3 Set-up

All the information below is also discussed in the documentation. Please refer to `AE.md` for more detailed instructions.

A.3.1 Installation

To start using CombiSan, clone its repository with:

```
git clone --recursive \
https://github.com/vusec/combisan.git
```

Then build two versions of LLVM, CombiSan's and an unmodified one to enable comparisons, with:

```
cd combisan
./build_llvm.sh
./build_llvm_clean.sh
```

Finally, set environment variables `CLEAN_LLVM_DIR` and `SPEC_DIR` to the paths of the newly-compiled, clean version of LLVM and to the SPEC installation, respectively.

A.3.2 Basic Test

To use COMBISAN, use the `clang(++)` binary of the newly built LLVM project, and add the `-fsanitize=address` compilation flag on any C(++) program. Since COMBISAN's code lives alongside ASan's inside the LLVM codebase, activating ASan's compilation flag enables COMBISAN's detection of both addressability issues and UUM errors. To also enable detection of UB bugs, add the desired flags (e.g., `-fsanitize=undefined` to detect all of them).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): COMBISAN simultaneously detects addressability issues, UUM errors, and UB bugs. It does so while matching the detection capabilities of ASan, MSan, and UBSan. This is proven by experiment (E1).

(C2): COMBISAN incurs low overhead compared to running sanitizer disjointly, as shown in experiment (E2).

(C3): COMBISAN integrates its detection capabilities while fuzzing. In particular, COMBISAN uses *deferred detection* of bugs to prevent early bugs from masking later ones. This is demonstrated by experiment (E3).

A.4.2 Experiments

(E1): [1 compute-hour]: Bugs detection accuracy.

How to: Run the tests from the Juliet Test Suite with COMBISAN, ASan, MSan, and UBSan.

Preparation: Compile COMBISAN's LLVM version with default configuration. This can be done using the `build_llvm.sh` script. Note that this script removes any other compiled version of COMBISAN's LLVM.

Execution: Execute the following command:

```
python3 setup.py run juliet csan_O0 \
--build --parallel=proc \
--parallelmax=$(nproc) \
--cwe 121 122 124 126 127 415 416 457 190 \
191 194 476 758 843 &> juliet_out
```

Notice how these tests are performed with `O0` as higher optimization levels (e.g., `O2`) mask bugs in Juliet. Additional tests can be made by changing the target tool from `csan_O0` to `asan_O0`, `msan_O0`, and `ubsan_O0`.

Results: The results can be filtered out with:

```
cat juliet_out | grep Passed
```

This reproduces the experiment from Section 6.2, which also discusses the minor differences in bug count.

(E2): [6 compute-hours]: Slowdown and memory overhead.

How to: Run the SPEC CPU benchmarks with COMBISAN, ASan, MSan, and UBSan.

Preparation: Compile COMBISAN's LLVM version with the *recovery* configuration detailed in Section 6.3. This can be done using the `build_llvm_SPEC.sh` script. This script removes any other compiled version of COMBISAN's LLVM.

Execution: Execute the following command:

```
python3 setup.py run spec2017 baseline_O2 \
asan_O2 csan-rec_O2 msan-rec_O2 \
--build --parallel=proc --parallelmax=1
```

Change `spec2017` to `spec2006` to change version.

Results: The results can be obtained with:

```
python3 ./setup.py report spec2017 \
results/last/ --aggregate geomean \
--field runtime:median maxrss:median
```

This reproduces the experiment from Section 6.3.

(E3): [1 compute-hour]: Fuzzing and deferred detection.

How to: Perform fuzzing with COMBISAN, and leverage deferred detection to handle multiple bug classes.

Preparation: Build COMBISAN's Docker image with:

```
cd docker/combisan/
docker build -t combisan .
```

Note that this needs to build COMBISAN's LLVM, so it may take some time. Then build a FTS subject's image with, e.g.:

```
cd docker/fts/libxml2/libxml2_combisan/
docker build -t libxml2_combisan .
```

Execution: Spawn a container and fuzz with:

```
docker run -it libxml2_combisan bash
$AFL/afl-fuzz -i in -o out -m none -- \
$EXECUTABLE_NAME_BASE
```

Note how the command to start fuzzing is generic and can be used on every target image we provide. For three projects (`libxml2`, `pcre2`, and `re2`), an additional seed directory (`crashes/`) is available to test fuzzing in the presence of bugs, showing deferred detection in action. The bugs present in this directory are two duplicates of a bug from each sanitizer (ASan, MSan, and UBSan).

Results: COMBISAN can successfully support fuzzing while detecting multiple classes of bugs. Further, deferred detection enables deeper fuzzing in case bugs of certain classes are detected, as demonstrated when fuzzing is started with duplicated crashing inputs: addressability issues always cause a crash, UUM errors trigger a slow path check on novelty, and UB bugs only result in a crash the first time they are encountered.

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/.