USENIX Security '26 Artifact Appendix: SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications

Oliver Braunsdorf[1], Tim Lange[1], Konrad Hohentanner[2], Julian Horsch[2], and Johannes Kinder[1]

[1]Ludwig-Maximilians-Universität München, Germany
[2]Fraunhofer AISEC, Germany

A Artifact Appendix

A.1 Abstract

This document describes the artifact accompanying the SafeFFI paper. This includes our modifications to rustc and LLVM, test sets, benchmarks and the corresponding scripts to build and execute them with SafeFFI, as well as our raw evaluation data. A more detailed listing of files can be found in `ArtifactContentsOverview.md`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact uses a VSCode devcontainer to build and test our custom rustc, which requires "seccomp=unconfined" privileges for running gdb with ASLR disabled inside the container. Additionally, 3rd party software is downloaded from Rust's package registry crates-io for testing SafeFFI. This 3rd party code and accompanied build scripts are executed during testing and benchmarking. While they could potentially contain malicious code that would be executed inside the unconfined container, it is unlikely because the downloaded crates are popular within the Rust community. However, users of the artifact should be aware of these security implications and might want to run the artifact inside a virtual machine for better isolation.

A.2.2 How to access

The artifact files are available on Zenodo: https://doi.org/10.5281/zenodo.17976648

A.2.3 Hardware dependencies

The artifact builds the Rust[1] compiler and LLVM[2] and thus requires a machine with sufficient RAM and disk space. We recommend at least 20 cores, 32 GB of RAM and 100 GB of free disk space. The artifact is supported to work under Linux on x86-64 and AARCH64 architectures. Because our experiments are run and compared on two different architectures, reproduction requires a machine with an x86-64 processor and a machine with an AARCH64 processor.

A.2.4 Software dependencies

All scripts are executed within docker containers and VSCode devcontainers[3]. Thus, the artifact only requires a build host with a modern Linux distribution that has docker and docker-compose installed (make sure the user on the build host is part of the docker-group on Linux), as well as Visual Studio Code (VSCode) on the evaluator's PC. For the build server, we used Ubuntu 24.04 on x86-64 and Fedora Linux Asahi Remix 39 on AARCH64 (MacStudioM2). We recommend the following commands to install the necessary dependencies for docker on Ubuntu:

```
sudo apt update
sudo apt install -y docker.io docker-compose
sudo usermod -aG docker $USER
newgrp docker
```

We recommend building and executing to be done on a remote build server with the above-mentioned hardware and software dependencies, connecting from the evaluator's PC via VSCode's remote development features.

[1] https://rustc-dev-guide.rust-lang.org/building/prerequisites.html?highlight=RAM#hardware
[2] https://llvm.org/docs/GettingStarted.html#hardware
[3] https://code.visualstudio.com/docs/devcontainers/containers

A.2.5 Benchmarks

No external benchmarks. All benchmarks used for the evaluation are contained in the artifact.

A.3 Set-up

To build the prototype, first, LLVM needs to be built, then rustc. Afterwards, the prototype can be tested. Note that detailed instructions and information about build flags can be found in the `README.md` in the artifact. However, for one-off building for artifact evaluation purposes, we provide basic commands for installation and testing below.

A.3.1 Installation

Prepare the host system (VM) Set entropy for ASLR to 28 bits, e.g. by using the following command as root on your host system (or inside a VM if you use one to isolate the artifact):

```
sudo sysctl -w vm.mmap_rnd_bits=28
```

> **Note**
>
> Please make sure to not skip setting the `vm.mmap_rnd_bits`. Due to a bug in the older versions of the vanilla LLVM sanitizers (https://github.com/google/sanitizers/issues/1614), this could otherwise lead to a false positive sanitizer detection which might affect reproducibility of our results.

Download and Patch LLVM and Rustc For a better overview of the changes, we provide our modifications to LLVM and rustc as git patches. To prepare the source code for building, run the following commands in the root of the artifact directory:

```
# go to root of directory
cd SafeFFI

# shallow clone and patch of llvm
git clone --depth 1 \
  https://github.com/rust-lang/llvm-project.git
cd llvm-project
git fetch --depth 1 \
  origin c3a26cbf6e73f2c5f8d03cee1f151d90a266ef3c
git checkout c3a26cbf6e73f2c5f8d03cee1f151d90a266ef3c
git apply --whitespace=nowarn ../llvm-patch.diff
cd ..

# shallow clone and patch of rust
git clone --depth 1 https://github.com/rust-lang/rust
cd rust
git fetch --depth 1 \
  origin 61edfd591cedff66fca639c02f66984f6271e5a6
git checkout 61edfd591cedff66fca639c02f66984f6271e5a6
git apply --whitespace=nowarn ../rust-patch.diff
cd ..
```

Build LLVM. Run the following commands in the root of the artifact directory:

```
cd SafeFFI/llvm-project
./local-build-with-docker.sh
```

Build rustc. For building rustc we provide a VSCode devcontainer.

- Open the directory (`SafeFFI/rust/`) in VSCode (via `File -> Open Folder`) and then press `F1 -> Dev Containers: Reopen in Container` to open the rust directory inside the VSCode devcontainer.
- Note: if VSCode seems to hang while building the devcontainer, pressing `F1 -> Reload Window` might help.
- When the container successfully spawned, open a new terminal inside VSCode. This should point you to the working directory `/workspaces/SafeFFI/SafeFFI` with the whole artifact mounted as bind-mount.
- For building our custom rustc **for the first time**, do:

  ```
  cd rust/SafeFFI-Docker/
  ./initial_build.sh
  touch /tmp/dealloc_functions.txt
  ```

- This produces our custom stage1 rust compiler that can be invoked with `cargo +stage1 <build command>`

> **Note**
>
> From here every other step for testing and evaluating happens inside the VSCode devcontainer.

Install python requirements Execute the following commands to install dependencies for running the artifact's python scripts:

```
python3 -m venv /home/vscode/safeffi-venv
source /home/vscode/safeffi-venv/bin/activate
pip install matplotlib pandas scipy
```

A.3.2 Basic Test

The `safeffi-tests` directory is also mounted in the devcontainer container at '/safeffi-tests'. You can use the tests to verify the SafeFFI Rust compiler.

```
cd /safeffi-tests/tests/rust/simple_sanitizer_tests
./safeffi_build.sh
```

The script runs two simple tests for out-of-bounds and use-after-free memory safety violations with SafeFFI on AddressSanitizer (ASan). They are succesful if the output contains `ERROR: AddressSanitizer: stack-buffer-overflow` and `ERROR: AddressSanitizer: heap-use-after-free`

Double-check `vm.mmap_rnd_bits` Running the following command in the docker container should print the `vm.mmap_rnd_bits` that you set earlier.

```
sudo sysctl vm.mmap_rnd_bits
```

A.3.3 Minimal Working Example

Below you can see the minimal working example of a command to build and run a program with SafeFFI and ASan. This command works in cargo-based build systems for Rust and mixed-language (including C/C++) code.

```
CC="/llvm-project/docker-install-dir/bin/clang" \
CFLAGS="-fsanitize=address" \
RUSTFLAGS="-Zsafeffi \
-Zsafeffi_temporal_mode=reinsert_checks_after_calls:/
    tmp/dealloc_functions.txt \
-Cllvm-args=-safeffi-nofree-out=/tmp/
    dealloc_functions.txt \
-Zsanitizer=address \
-Clinker=/llvm-project/docker-install-dir/bin/clang \
-Clink-arg=-fuse-ld=lld" \
cargo +stage1 run \
--target=aarch64-unknown-linux-gnu
```

We explain the flags in more detail here. The first two flags are for the C/C++ compiler.

- `CC="/llvm-project/docker-install-dir/bin/clang"`: specify the clang compiler compatible with our LLVM toolchain.
- `CFLAGS="-fsanitize=[address|hwaddress]"`: enable the sanitizer for C/C++ code.

The remaining flags inside of `RUSTFLAGS` are consumed by the Rust compiler.

- `Zsafeffi`: activate SafeFFI for Rust source code (this implicitly adds `-hwasan_safeffi_propagation_elision` to the rustc's invokation of LLVM's sanitizers to activate the SafeFFI-Lib)
- `Zsafeffi_temporal_mode`: activate(`reinsert_checks_after_calls`) or deactivate (`reinsert_none`) SafeFFI's additional checks for Free-During-Scope vulnerabilities. For reinserting checks you need to specify a file that should exist and be empty: `reinsert_checks_after_calls=</path/to/noFreefunctions>`
  This is where our on-the-fly callgraph analysis writes the functions that are classified using our NoFree SCC Pass as descibed in §6.4 of the paper.
- `-Cllvm-args=-safeffi-nofree-out` has to point the the same file if the second option of the previous flag is used.
- `-Zsanitizer=[address|hwaddress]` : selects the underlying sanitizer for Rust (this has to match the sanitizer for C++). If Hardware-assisted AddressSanitizer (HWASan) is selected an additional `-Ctarget-feature=+tagged-globals` is necessary.
  `-Clinker=/llvm-project/docker-install-dir/bin/clang` :
  specify the clang linker compatible with our LLVM toolchain.
- `-Clink-arg=-fuse-ld=lld` : specify the use of the lld linker that is compatible with our LLVM toolchain.
- `cargo +stage1 run`: invokes SafeFFI's custom stage1 rust compiler to build and run the program.
- `--target=[x86_64|aarch64]-unknown-linux-gnu` : necessary for Rust to sanitize the Rust application but not the `build.rs` scripts.

We included an example of a mixed-language application that can be built with this command in the directory `/safeffi-tests/tests/ffi/minimal_working_example`. It contains an out-of-bounds vulnerability that is detected by SafeFFI's inserted checks when running this example with the above command.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SafeFFI correctly detects all memory safety violations that the underlying sanitizer detects in a set of known real-world memory safety vulnerabilities. This is proven by experiment E1 which is described in Section 7.2 of the paper whose results are reported in Table 1.

(C2): SafeFFI effectively reduces the number of sanitizer checks compared to the instrumentation with vanilla ASan. This is proven by experiment E2 which is described in Section 7.3 "Elided Checks" of the paper whose results are illustrated in Figure 5.

(C3): SafeFFI reduces the run-time overhead of ASan and HWASan. This is proven by experiment E2 which is described in Section 7.3 Paragraphs "Run-time Performance" and "ARM and HWASan" of the paper whose results are illustrated in Figures 6 and 7.

(C4): SafeFFI achieves superior compile-time overhead compared to state-of-the-art approaches. This is proven by experiment E2 which is described in Section 7.5 of the paper whose results are illustrated in Figure 8.

(C5): SafeFFI works robustly in Mixed-Language Application (MLA) scenarios where Rust and C/C++ interoperate via FFI and does not produce false positives nor false negatives. This is proven by experiment E3 which is described in Section 7.4 of the paper.

A.4.2 Experiments

The experiments described below describe how to reproduce the major claims about SafeFFI made in the paper. We do not include reproduction of test results from related work (ERASan and RustSan) as these are outside the scope of our artifact.

To reproduce results for SafeFFI on HWASan, the experiments need to be executed on an AARCH64 machine since HWASan is only supported for that architecture. For ASan, the experiments can be executed on both x86-64 and AARCH64 machines. Table 1 summarizes the exact environment we used for our experiments.

Please don't execute the experiments in parallel on the same host as this could lead to resource contention and unreliable results.

(E1): Correctness [10 human-minutes + 60 compute-minutes]: Run a set of known real-world memory safety vulnerability PoCs compiled with SafeFFI and check that all memory safety violations are detected.

Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3.

Execution: Inside the VSCode devcontainer, run:

```
cd /safeffi-tests/tests/rust
./test-all-pocs-with-safeffi.py
```

Results: The results are printed to the console in a summarizing table that shows the detection results for each PoC and each configuration of sanitizer configuration. The result classification for vanilla sanitizers is `DETECTED` or `FAILED_TO_DETECT`. For SafeFFI, the additional classifications `cast-check` and `temporal-check` indicate SafeFFI-specific detections. The results should match Table 1 in the paper. As indicated in the paper: "Some vulnerabilities in the dataset are non-deterministic which makes it necessary to run each compiled test binary multiple times to observe the vulnerability reliably, especially for HWASan because its probabilistic detection mechanism adds further non-determinism on top, even for deterministic vulnerabilities. If at least one run triggered the vulnerability, we classified it as detected." Thus, the script executes each compiled PoC binary 100 times and the result output shows the number of executions per PoC in which the sanitizer failed to detect the vulnerability. For HWASan-based configurations, we have seen failure rates of up to ∼10 times out of 100 executions.

(E2): Check Elision and Performance Benchmarks [30 human-minutes + 10 compute-hours]: Run a set of benchmarks to measure the number of sanitizer checks elided by SafeFFI, as well as compile-time overhead and runtime overhead compared to vanilla ASan/HWASan and a baseline without instrumentation.

Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3. Initialize the frozen crates-io index as git repository:

```
cd /playground
rm -rf \
  crates.io-index-archive-snapshot-2020-11-20/
git clone \
  https://github.com/SafeFFI/safeffi-crates-io-
    index.git \
  crates.io-index-archive-snapshot-2020-11-20
```

To check if the benchmarks can be correctly downloaded and run in your setup, you can first execute:

```
cd /playground/rust-benchmarking-scripts
./run.py --smoke
```

This executes a small subset of the benchmarks to verify that everything is working correctly(ca. 20-30 minutes on X86, ca. 40-60 minutes on AARCH64). You should see the script terminating without errors and produce the same CSV files as listed below.

Execution: Inside the VSCode devcontainer, run:

```
cd /playground/rust-benchmarking-scripts
./run.py
```

> **Note**
>
> Some of the benchmarks are very long-running, especially if instrumented with non-optimized sanitizers. We set a timeout for each individual benchmark execution to 120 minutes which should be sufficient to finish all benchmarks. You might want to adjust the threshold variable `BENCH_TIMEOUT` in `run.py:22` to make the benchmarking process either faster or more robust against timeouts on your evaluation hardware. During Usenix Artifact Evaluation, four benchmarks (Chrono, Adler, Base64, and Image) failed in the vanilla ASan and SafeFFI-ASan versions on some systems. Only those four benchmarks are using the Criterion crate as a benchmarking backend. The failure pattern indicates that the common reason is a bug in Criterion (we used it's older version 0.3.2) which triggers ASan[a].
>
> [a] Criterion's custom `black_box()` has the same issue as: https://github.com/rust-lang/rust/pull/104110/

Results: The results are stored as CSV files in `/playground/rust-benchmarking-scripts/tmp` and contain the absolute numbers for the measurements that resemble the results reported in the paper.

- `check_elision_stats.csv` (Claim C2)
- `runtime_stats.csv` (Claim C3)
- `compilation_stats.csv` (Claim C4)

Table 1: The exact environment we used for our experiments.

Architecture	CPU	RAM	Host-OS	Container-OS
x86-64	AMD EPYC 9645 (384 cores)	1.5 TB	Ubuntu 24.04	Ubuntu 20.04
AARCH64	Apple M2 Ultra (24 cores)	192 GB	Asahi Linux 6.12.0	Ubuntu 20.04

(E3): Robustness in MLA Scenarios [10 human-minutes + 10 compute-minutes]: Run a systematically created set of vulnerable and benign test cases of mixed-language applications with SafeFFI and check that the test cases are classified correctly. These tests require HWASan and therefore need to be run on an AARCH64 machine.
Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3.
Execution: Inside the VSCode devcontainer, run:

```
cd /playground/safeffi-systematic-tests/
./generation.py
```

Results: The results are printed to the console in a summarizing table that shows the detection results for each test case. The output is in Markdown table format. The test is successful if all test cases contain the string *"correctly"* in the column *Error Detected*. Log files for each individual test case are stored in the `log/` subdirectory.

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/.