# Provably-Safe Multilingual Software Sandboxing using WebAssembly

Jay Bosamiya, Wen Shih Lim, and Bryan Parno, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

## A Artifact Appendix

### A.1 Abstract

Our artifact contains the source files, scripts, and other necessary files for reproducing the results described in the paper. It consists of the two compilers (vWasm and rWasm), the semantics fuzzer, as well as benchmarking scripts. To run these, one needs a Linux x86-64 machine, or a Docker environment capable of running it. Since one of the two compilers, namely vWasm, contains a machine-checked proof, the artifact also contains instructions to re-verify that all parts of the proof are indeed accepted by F*.

### A.2 Artifact check-list (meta-information)

- **Program:**
  - vWasm: a formally-verified provably-safe sandboxing compiler, built in F*
  - rWasm: a high-performance informally-verified provably-safe sandboxing compiler
  - wasm-semantics-fuzzer: a tool for providing greater assurance in the semantic correctness of any Wasm implementation
- **Compilation:** vWasm requires F*, OCaml, nasm, etc.; the rest require a Rust installation. We include a Docker image with all requirements in the artifact.
- **Data set:** Benchmarks and micro-benchmarks are included in the artifact. See `benchmarks/`, `microbenchmark-compare-read-arr/` and `image-conversion-scenario/` in the main repository.
- **Run-time environment:** Our artifact was developed and tested on recent Linux-based systems. We include a Docker image with all requirements.
- **Hardware:** Requires an x86-64 machine. We tested on an AMD Ryzen 3700x (64GB memory) and on an Intel i9-9900K (128GB memory). While almost everything *should* run on a machine with less memory, we recommend 32GB or higher to allow parallelism to save user time in some of the memory-intensive steps.
- **Output:** We provide more detail in the artifact `README.md` files, but in short, building vWasm will verify and compile the vWasm compiler, building rWasm will compile the rWasm compiler, building wasm-semantics-fuzzer will build the fuzzer, and running the benchmarks will use the compilers to run the experiments described in the paper.
- **How much disk space required (approximately)?:** Approximately 5 GB for the Docker image; the rest of the files are negligible in size. When running benchmarks, space usage can increase a lot more, and thus it is best to have free space on the order of approximately 100 GB.
- **How much time is needed to prepare workflow (approximately)?:** The provided Docker image contains all requirements for the two compilers, and loading it from the exported image should only take a minute or two. Building the Docker from scratch takes significantly longer (an hour or two). The other Wasm runtimes being benchmarked against are not all included in the Docker image, but instructions are included and should not take more than 15 minutes to get running.
- **How much time is needed to complete experiments (approximately)?:** Re-verifying vWasm, and running the main execution-time benchmarks are the most time consuming parts of the artifact. In total, expect this to take multiple hours, with times varying depending on the available parallelism on the machine being tested on.
- **Publicly available:** The latest version of the repositories:
  - `https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22` (top-level repository that contains the benchmarks, and imports the rest as git submodules)
  - `https://github.com/secure-foundations/vWasm`
  - `https://github.com/secure-foundations/rWasm`
  - `https://github.com/secure-foundations/wasm-semantics-fuzzer`

  The first of the above links contains the other three as git submodules, pinned to specific git commits.
- **Code licenses:** BSD 3-Clause License
- **Archived (stable reference):** Top-level repository, with the other repositories fixed to specific git commit hashes: `https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22/tree/6f5668d3f216aeef65cf2bf2d916a40d3c750e53`

### A.3 Description

#### A.3.1 How to access

`https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22` is a link to the top-level artifact repository, which contains the rest of the related repositories as git submodules. To get them all in one single command, run `git clone --recursive https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22`

Instructions for obtaining the vWasm Docker image can be found at `https://github.com/secure-foundations/vWasm/tree/main/.docker`, and a top-level `Dockerfile` can be found at the root of the top-level repository.

### A.3.2 Hardware dependencies

Requires an x86-64 machine. 32+ GB of memory is recommended.

### A.3.3 Software dependencies

Requires Docker installed, preferably on a Linux host. All other requirements work inside the container.

### A.3.4 Data sets

Included in the artifact.

### A.3.5 Models

N/A

### A.3.6 Security, privacy, and ethical concerns

N/A

## A.4 Installation

We provide detailed instructions throughout the artifact in the form of `README.md` files.

In short, install Docker (`https://www.docker.com/get-started`), recursive-clone the repositories (`git clone --recursive ...` command from above), go to `provably-safe-sandboxing-wasm-usenix22/vWasm/.docker` and follow instructions there to download and import the pre-built image, and then jump inside the provided Docker container. Following this, everything else can be run inside the container.

## A.5 Experiment workflow

For each experiment, we provide a relevant `README.md` file with detailed instructions. We recommend executing steps in the following order:

1. Build the Docker image, and jump into the container.
2. Run the verification and build process for vWasm.
3. Run the build process for rWasm.
4. Generate `.wasm` files using wasm-semantics-fuzzer and run them in vWasm and rWasm.
5. Run the microbenchmark.
6. Run the execution-time benchmarks in the `benchmarks/` directory (here, you can choose to compare against all other tools, or you can run fewer tools, see the `README.md` for instructions).
7. Run the image-conversion-scenario.

## A.6 Evaluation and expected results

Our paper claims to make the following contributions (copied verbatim from Section 1):

1. An exploration of two distinct techniques to achieve provably safe, performant, multi-lingual sandboxing. We implement these as open-source tools, and evaluate them on a collection of quantitative and qualitative metrics.
2. vWasm, the first verified sandboxing compiler for Wasm, achieved via traditional machine-checked proofs.
3. rWasm, the first provably safe sandboxing compiler with competitive run-time performance. We achieve this using non-traditional repurposing of existing tools to provide provable guarantees without writing formal proofs.

We provide detailed instructions throughout the artifact in the form of `README.md` files.

To confirm that vWasm is formally verified, execute the steps in `vWasm/README.md`. Each file in the project is machine-checked and only once all files are verified by F*, will it produce the extracted OCaml files, which are then compiled to an executable compiler. The high-level theorem statement being proven can be found in `vWasm/compiler/sandbox/Compiler.Sandbox.fsti`.

Both vWasm and rWasm can be run independently to compile any Wasm module. Built-in runtime support is provided for Wasm modules that expect a WASI interface.

Using wasm-semantics-fuzzer, one can perform validation checks that the semantics implemented by vWasm and rWasm do indeed match expected Wasm semantics.

The image-conversion-scenario demonstrates a converter from GIFs to JPEGs, using version of libraries susceptible to CVE-2008-0554. Using `./see_cve_impacts.sh`, one can run a proof-of-concept input that demonstrates how the native, vWasm-built, and rWasm-built versions of the programs perform. Expected behavior is detailed further in the script before each execution, but in summary, the native version (not protected by vWasm or rWasm) will suffer a bad crash, while the vWasm and rWasm versions will successfully safely trap the violation.

Quantitative evaluation is performed using the benchmarks and provided scripts. For execution time and run time benchmarks, results should be within the error bars, if run on a similarly modern hardware. The microbenchmarks are more susceptible choice of hardware (as shown in Figure 8), and we have tested only on an AMD Ryzen 3700x and an i9-9900k.

## A.7 Experiment customization

After vWasm and rWasm are built, you can test them with any WASI-enabled modules you like. Instructions are provided in the relevant `README.md` files.

For the execution-time benchmarks, not all competing Wasm execution runtimes are included in the vWasm Dockerfile, but the top-level Dockerfile does include them all. Additionally instructions for how to install them, or how to selectively disable the runtimes are given.

## A.8 Version

Based on the LaTeX template for Artifact Evaluation V20220119.