USENIX Security '25 Artifact Appendix: Encarsia: Evaluating CPU Fuzzers via Automatic Bug Injection

Matej Bölcskei
ETH Zurich

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

A Artifact Appendix

A.1 Abstract

We provide the source code of ENCARSIA along with a Docker-based evaluation setup to facilitate reproducing the results presented in our accompanying paper and to support ENCARSIA's deployment for evaluating future CPU fuzzers. These artifacts demonstrate that ENCARSIA is a fully functional tool capable of injecting bugs into CPUs, formally verifying their architectural observability, and leveraging the generated bugs to assess fuzzers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

ENCARSIA is a bug injection and fuzzer benchmarking tool that does not pose any security risks, as it is not designed to attack the evaluation system. However, its high degree of parallelism, combined with the significant computational cost of formal verification and RTL simulation, can overwhelm system resources and potentially cause crashes.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at `https://doi.org/10.5281/zenodo.14664723`. Alternatively, they can be accessed via GitHub at `https://github.com/comsec-group/encarsia`.

A.2.3 Hardware dependencies

ENCARSIA runs on any standard hardware, but requires substantial computing resources to handle the demands of formal verification, RTL simulation, and to take full advantage of parallelization for accelerating the experiments. We therefore recommend a machine with at least 32 CPU cores, 256 GB of main memory, and 512 GB of disk space. For detailed requirements of each experiment, refer to their descriptions in Section A.4.2.

A.2.4 Software dependencies

The following software dependencies are required for running ENCARSIA:

- **`docker`**: to run the experiments in the provided Docker environment.
- **`make`**: for automating tasks like building the Docker image.
- **`tar`**: to extract the EnCorpus dataset, which is provided as a .tar.gz archive.
- **`python3`**: for parsing the results of our bug survey.

If you prefer to run the experiments outside the Docker setup, the Dockerfile can serve as a reference for the necessary software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone the artifact repository using the GitHub link provided in Section A.2.2, or download it from the permanent access link to our artifacts on Zenodo. After cloning the repository, navigate to the root directory of the repository and run `make pull` to fetch the pre-built Docker image. Alternatively, you can run `make build` to build the Docker image locally.

After obtaining the Docker image, use `make run` to start a container from the image. Note the container ID displayed in the terminal output. You can use it later to restart and attach to the container for further experiments with `docker start <container_id> && docker attach <container_id>`.

A.3.2 Basic Test

Make sure the Docker container is running and attached. Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p 30` to confirm that ENCARSIA is functioning correctly and parsing the EnCorpus dataset as expected.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Bugs in open-source CPUs often stem from two simple syntactic transformations: mix-ups of signals or logic expressions and errors in conditional statements. This is proven by the bug survey **(E1)** described in Section 4 of the paper whose results are reported in Table 4.

(C2): ENCARSIA can rapidly inject a large number of diverse, realistic bugs into CPUs of varying complexity and design paradigms. This is proven by the injection experiment **(E2)** described in Section 7.1 of the paper whose results are reported in Table 5.

(C3): ENCARSIA can formally prove the architectural observability of bugs within a practical timeframe using computational resources typical for hardware design and verification. This is demonstrated by experiment **(E3)**, which follows a similar approach to the verification experiment described in Section 7.1 of the paper whose results are reported in Table 6. To make our verification framework more accessible, we provide a fully open-source version based on Yosys.

(C4): Instruction-granular bug detection mechanisms do not demonstrate greater potential for detecting bugs. This is demonstrated by the instruction-granular bug detection evaluation **(E4)** described in Section 8.1 of the paper whose results are reported in Table 8.

(C5): The hardware-specific structural coverage metrics, advertised as central by many fuzzers, are of little help in detecting bugs. This is demonstrated by the coverage metrics evaluation **(E5)** described in Section 8.2 of the paper whose results are reported in Table 9.

(C6): The fuzzer seeds are a key factor that determines which bugs will eventually be detected. This is demonstrated by the seed program experiment **(E6)** described in Section 8.3 of the paper whose results are reported in Table 10.

A.4.2 Experiments

All experiments are intended to be run within the provided Docker environment to ensure consistency and reproducibility. Before running each experiment, ensure that your system has sufficient computing resources available to meet the requirements specified for the experiment.

(E1): [Survey (Section 4)] [5 human-minutes]: The results of our bug survey are available in `survey/classification/synthetic.json` and `survey/classification/natural.json` within the main artifacts repository (not Docker image).
Execution: Use `survey/classification/plot.py` to display the results of the survey.
Results: The results of the survey confirm that all identified observable bugs indeed do fall into one of the two categories.

(E2): [Injection (Section 7.1)] [5 human-minutes + 20 compute-minutes + 100GB disk + 8GB memory]: This experiment exactly replicates the injection experiment described in Section 7.1 of the paper.
Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/Injection -H ibex rocket -p 30`. Optionally, add `boom` to the `-H` option to inject bugs into BOOM, but note that this requires up to 512GB of disk space.
Results: This experiment injects around 1000 Signal Mix-ups and 1000 Broken Conditionals per CPU. The resulting `host.v` files and injection logs can be found in the experiment directory at `/encarsia-meta/out/Injection`. A summary of the injection results, similar to Table 5 in the paper, is printed to the terminal. We expect the summary table to closely match the one presented in the paper.

(E3): [Bug Verification] [5 human-minutes + 1 compute-hour + 6GB disk + per-process memory (device dependent: 4 GB for Ibex, 8 GB for Rocket, 32 GB for BOOM)]: This experiment closely replicates the verification experiment outlined in Section 7.1 of the paper, with the main difference being our use of a fully open-source verification setup based on Yosys. Additionally, we limit the experiment to bugs from the EnCorpus bug set to reduce the duration of the experiment.
Execution: Assess your system's computing resources to determine the number of parallel processes it can support for this experiment. Then, navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H ibex rocket boom -p NUM_PROC -Y`, replacing `NUM_PROC` with the number of parallel processes. You can also execute the experiment on one device at a time by running `python encarsia.py -d out/EnCorpus -H DEVICE -p NUM_PROC -Y` for each of the three devices. This ensures optimal memory usage, despite varying memory requirements across devices.
Results: This experiment generates formal proofs of architectural observability for the EnCorpus bugs using the Yosys setup. The resulting verification log (`yosys_verify.log`) and proof of observability (`yosys_proof.S`) can be found in the EnCorpus experiment directory at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the verification results, similar to Table 6 in the paper, is printed directly to the terminal. Note that EnCorpus was verified using the JasperGold setup, which is more robust and powerful. As a result, not all EnCorpus bugs are expected to verify successfully using Yosys. We nevertheless expect Yosys to verify most of the EnCorpus bugs in the simpler CPUs (Ibex and Rocket) and a smaller subset in the more complex BOOM. Furthermore, we expect the average verification time per bug to be similar to the values

reported in Table 6 of the paper.

(E4): [Instruction-granular bug detection evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.1 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F no_cov_difuzzrtl no_cov_processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 8 in the paper, is printed directly to the terminal.

We expect the results to match those presented in the paper, except for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14. These bugs are detected by DifuzzRTL and Processorfuzz in the Docker setup, but remain undetected in the bare-metal setup used to generate the data presented in the paper despite several fuzzing re-runs. We suspect this discrepancy stems from differing versions of dependencies, such as Spike or Verilator, used internally by the fuzzers. However, the lack of transparency regarding which tools are used and their specific versions makes it difficult to determine the exact cause. Despite this discrepancy, we firmly believe that the major claims of the paper remain valid.

(E5): [Coverage metrics evaluation] [5 human-minutes + 10 compute-hours + 100GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.2 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl processorfuzz`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 9 in the paper, is printed directly to the terminal.

As in the previous experiment, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14.

(E6): [Seed program evaluation] [5 human-minutes + 90 compute-minutes + 10GB disk + 4GB memory per parallel process]: This experiment closely replicates the fuzzing experiment outlined in Section 8.3 of the paper, with the main difference being the reduced fuzzing duration of 30 minutes.

Note that this experiment reuses the results of the DifuzzRTL evaluation from experiment **(E5)**, if available, which reduces computation time and disk space requirements. If the results of experiment **(E5)** are not available, the requirements are approximately half those of experiment **(E5)**.

Execution: Navigate to the `/encarsia-meta` directory and run `python encarsia.py -d out/EnCorpus -H rocket boom -p 30 -F difuzzrtl cascade`.

Results: This experiment generates two key outputs: a fuzzing log stored in `fuzz.log`, and the bug detection results (after filtering out false positives) in `check_summary.log`. Both files are located within the corresponding fuzzer directories at `/encarsia-meta/out/EnCorpus`. Additionally, a summary of the fuzzing results, similar to Table 10 in the paper, is printed directly to the terminal.

As in the previous experiments, we observe discrepancies between the Docker and bare-metal setups for Rocket Signal Mix-up 1 and BOOM Signal Mix-ups 9 and 14 on DifuzzRTL. However, the results for Cascade exactly match those reported in the paper.

A.5 Troubleshooting Guide

- **OOM Errors:** Ensure your system meets the minimum memory requirements specified for the experiment by reducing the number of parallel processes (`-p NUM_PROC`).
- **Running Out of Disk Space:** Verify that your system has sufficient disk space before starting an experiment. Clean up old experiment outputs if necessary.
- **Badly Terminated Parallel Experiments:** If an experiment is terminated ungracefully, some processes may remain running in the background. Use `ps` to identify and terminate any processes stuck in an infinite loop.

A.6 Notes on Reusability

ENCARSIA can be easily extended to support additional CPUs by creating a new `EncarsiaConfig` instance in `/encarsia-meta/config.py`. For more information, see the README in the ENCARSIA repository.

A.7 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/.