Efficient Symbolic Execution of Software under Fault Attacks (Artifact)

Yuzhou Fang
University of Southern California, Los Angeles, CA, USA

Chenyu Zhou
University of Southern California, Los Angeles, CA, USA

Jingbo Wang
Purdue University, West Lafayette, IN, USA

Chao Wang
University of Southern California, Los Angeles, CA, USA

Abstract

We propose a symbolic execution method for analyzing the safety of software under fault attacks both accurately and efficiently. Fault attacks leverage physically injected hardware faults in an embedded system to break the safety of a software program. While there are existing methods for analyzing the impact of maliciously injected hardware faults on the embedded software, they suffer from inaccurate fault modeling and inefficient fault analysis. To overcome these limitations, we propose two novel techniques. First, we propose a new fault modeling technique that leverages automated program transformation to add symbolic variables to the original program, to accurately model the new program behavior induced by the injected faults. This new fault modeling approach has two advantages over existing techniques: (a) the fault-induced program behavior is closely related to what attackers exploit in practice and (b) the automatically transformed program may be analyzed by any downstream fault analysis algorithm. Second, we propose an efficient symbolic execution algorithm that is designed specifically for conducting fault analysis on the transformed program. It leverages two pruning techniques to mitigate path explosion, which is the main performance bottleneck of symbolic execution in general and, in this particular application, is exacerbated by the additional fault-induced program behavior. We have implemented the proposed method and evaluated it on a variety of benchmark programs. The experimental results show that our method significantly outperforms the state-of-the-art techniques. Specifically, our method not only drastically reduces the overall running time of symbolic execution but also retains its error detection capabilities. Compared to the current state-of-the-art, it is able to detect previously-missed safety violations and at the same time avoid bogus violations. Furthermore, compared to the baseline algorithm, our optimized symbolic execution algorithm can be orders-of-magnitude faster.

2012 ACM Subject Classification Theory of computation → Program verification; Software and its engineering → Software testing and debugging

Keywords and phrases Symbolic Execution, Safety Verification, Fault Attack, Embedded Software

Digital Object Identifier 10.4230/DARTS.12.1.10

Related Article Yuzhou Fang, Chenyu Zhou, Jingbo Wang, and Chao Wang, "Efficient Symbolic Execution of Software Under Fault Attacks", in 40th European Conference on Object-Oriented Programming (ECOOP 2026), LIPIcs, Vol. 372, pp. 4:1–4:25, 2026. https://doi.org/10.4230/LIPIcs.ECOOP.2026.4

Related Conference 40th European Conference on Object-Oriented Programming (ECOOP 2026), June 29–July 3, 2026, Brussels, Belgium

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2026 Call for Artifacts and the ACM Artifact Review and Badging Policy.

Dagstuhl Artifacts Series
Schloss Dagstuhl – Leibniz-Zentrum für Informatik,
Dagstuhl Publishing, Germany

1 Scope

The artifact is intended to support the following claims of the paper:

- Our new fault modeling technique can reveal *previously-unknown* safety violations that are missed by existing techniques. The claims are made in **Section 6.3**, by **Tables 3 and 4**.
- Our new *weakest precondition* based pruning technique can effectively mitigate the path explosion problem when applying symbolic execution to fault analysis. The claims are made in **Section 6.4**, by **Figure 7**.

2 Content

The artifact is a Docker container that carries all the necessary dependencies of the tool, source code of the implementation, and the benchmark programs used in the paper. Specifically, it includes the following contents:

- The source code of our implementation, including:
  - An LLVM-based program transformation pass that models the fault-induced program behaviors by adding symbolic variables to the original program.
  - A customized symbolic execution engine based on KLEE, which supports the two pruning techniques proposed in the paper; one is *fault saturation* and the other is *weakest precondition* based pruning.
- The compiled and ready-to-go binaries of our implementation, which can be directly used to reproduce the experimental results in the paper.
- The benchmark programs used in the paper, including the ones from VerifyPIN test suite and the ones from SV-COMP.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The artifact is also available on Zenodo at: `https://zenodo.org/records/19560319`.

In addition, the artifact is also available as a Docker image on DockerHub at: `https://hub.docker.com/r/yuzhouf/ecoop46`.

The artifact is packaged as a Docker container with all necessary dependencies pre-installed and precompiled binaries ready to go. To set up the artifact, follow the steps below.

1. **Obtain the Docker image.** Pull the image from DockerHub:

   ```
   docker pull yuzhouf/ecoop46:latest
   ```

   Alternatively, build from the Dockerfile included in the artifact:

   ```
   docker build -t yuzhouf/ecoop46:latest .
   ```

2. **Create and start the container.** First, unzip the artifact archive and navigate into the extracted artifact folder:

   ```
   unzip ecoop46.zip
   cd ecoop46
   ```

   Then, create and start the container by mounting the `ecoop46` subfolder found inside the artifact folder:

```
docker run -d --name ecoop46 \
    -v $(pwd)/ecoop46:/home/ecoop46 \
    yuzhouf/ecoop46:latest
```

Note that you must mount the `ecoop46` subfolder (not the artifact root itself), as mounting the root directory would cause path-related issues inside the container.

3. **Log into the container.**

```
docker exec -it -u ecoop46 ecoop46 bash
```

The username and password are both `ecoop46`. After logging in, verify that the `ecoop46` user owns its home directory:

```
ls -ld /home/ecoop46
```

The output should show `ecoop46` as the owner. If instead it is owned by `root` (which can happen due to volume mounting), exit the container and fix the ownership as root:

```
docker exec -it ecoop46 bash
chown -R ecoop46:ecoop46 /home/ecoop46
exit
```

Then log back in as `ecoop46`.

4. **Navigate to the project directory.** Once inside the container, change to the project directory:

```
cd ~/code/fpass
```

A quick smoke test is available at `~/code/fpass/experiment/smoke-test`. See `fpass/README.md` inside the container for detailed instructions on reproducing all evaluation results.

Note that more detailed instructions can be found after unzip the artifact, in the `README.md` file included in the artifact.

4 Tested platforms

We conducted the experiments on a computer with a 4.7 GHz AMD R9 7900X CPU and 32 GB of RAM, running the Ubuntu 22.04 LTS Linux system. The artifact is distributed as a Docker container and is expected to run on any x86-64 machine with Docker installed and a comparable amount of memory. Reproducing the full experimental results requires a 90-minute per-program timeout, as used in the paper.

Important note on reproduction time and platform variability. Reproducing the *complete* set of experiments reported in the paper is an extremely time-consuming process, potentially taking **many hours to days** depending on the hardware configuration. Furthermore, because symbolic execution is sensitive to low-level factors such as CPU architecture, memory subsystem performance, and solver heuristics, the absolute execution times may vary significantly across different platforms. While the *qualitative* results (e.g., which violations are found and the relative effectiveness of the pruning techniques) are expected to be consistent, the exact execution times reported in the paper should be interpreted as reference values obtained on our specific hardware setup rather than as precise, universally reproducible numbers. We recommend using the provided smoke test for a quick validation before attempting a full reproduction.

5 License

6 MD5 sum of the artifact

76ceeaa2a8bd59aef9eaeb19de9cdb1f

7 Size of the artifact

129.7 MB