USENIX Security '24 Artifact Appendix: Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolized Stateful Fuzzing

Yibo Wang
Syracuse University
ywang349@syr.edu

Yuzhe Tang
Syracuse University
ytang100@syr.edu

Kai Li
San Diego State University
kli5@sdsu.edu

Wanning Ding
Syracuse University
wding04@syr.edu

Zhihua Yang
Syracuse University
zyang47@syr.edu

A Artifact Appendix

A.1 Abstract

This artifact presents the design and implementation of MPFUZZ, a symbolized-stateful mempool fuzzer that automatically identifies asymmetric DoS vulnerabilities in an Ethereum mempool. MPFUZZ defines the search space using mempool states covered by symbolized transaction sequences. It efficiently explores this space by utilizing feedback from symbolized state coverage and evaluating the potential of intermediate states to trigger bug oracles. The artifact demonstrates that MPFUZZ discovers new asymmetric DoS vulnerabilities in the Ethereum client and shows the process for identifying these vulnerabilities.

A.2 Description & Requirements

The artifact is a symbolized-stateful Ethereum mempool fuzzer designed to automatically identify asymmetric DoS vulnerabilities. MPFUZZ operates on a reduced version of the mempool under test (MUT) deployed locally. The fuzzing process facilitated by MPFUZZ uncovers short exploit sequences that can subsequently be developed into comprehensive, actionable exploits.

The artifact includes a modified Go-Ethereum client based on Geth v1.10.11, a prototype of MPFUZZ implemented in Python, and associated dependencies. The source code of the modified Go-Ethereum client is provided in our artifact and can be built from source. The MPFUZZ prototype necessitates Python 3.9 and several Python libraries, specifically web3, numpy, pandas, and graphviz.

A.2.1 Security, privacy, and ethical concerns

Our artifact contains many pre-generated public-private key pairs. It is imperative that these keys are not used on real networks to avoid potential financial loss. Utilizing these keys in a live environment poses significant security risks, as they may be easily compromised. Evaluators must ensure that these keys are restricted to controlled, test environments.

A.2.2 How to access

To access our artifact, please use a web browser to navigate to the following URL: `https://doi.org/10.6084/m9.figshare.26068909.v6`. Download all the files available in the root folder. After downloading all the files from the Figshare project, unzip the `key_new.zip` file and ensure that the unzipped folder retains the same name, `./key_new/`.

A.2.3 Hardware dependencies

To evaluate our artifact, no specific CPU hardware is required. While our artifact does not have a minimum requirements for memory and storage, higher capacities are advantageous for improving overall performance.

A.2.4 Software dependencies

To properly evaluate our artifact, specific operating system and software packages are required. Our artifact is designed to operate on Unix-like operating systems, such as Ubuntu. Additionally, Python version 3.9 or later must be installed, serving as the foundational environment for our artifact's execution. A few Python libraries are essential to its operation include web3, numpy, pandas, and graphviz.

A.2.5 Benchmarks

None

A.3 Set-up

In this section, we provide instructions for setting up the environment and running our artifact for evaluation purposes. After downloading all the files from the Figshare project and unzipping `key_new.zip`, it is essential to build the modified Go-Ethereum client from source. To do so, it requries first unzip the go-ethereum-1.10.11.zip, keep the unzipped folder with the same name as ./go-ethereum-1.10.11/. Then, build the author-modified version of Go-Ethereum client via the following command:

```
$ make -C go-ethereum-1.10.11
$ cp go-ethereum-1.10.11/build/bin/geth ./geth
```

Next, make the start Ethereum script executable via the following command:

```
$ chmod +x ./start_ethereum.sh

$ chmod +x ./start_ethereum_e2a.sh

$ chmod +x ./start_ethereum_e2b.sh
```

A.3.1 Installation

To run our MPFUZZ prototype implemented in Python, it is essential to first install Python 3.9 by following the installation instructions provided on the Python website: https://www.python.org/downloads/.

Next, install the required Python libraries using the pip command:

```
$ pip3 install web3 numpy pandas graphviz
```

This command installs the necessary dependencies, namely web3 for Ethereum interaction, numpy for numerical computations, pandas for data manipulation, and graphviz for graph visualization tasks.

A.3.2 Basic Test

We conduct two tests: first, to ensure the Ethereum client is operational, and second, to verify the readiness of the environment for running the MPFUZZ Python prototype.

To test the Ethereum client, execute the following command in the current directory to start the client:

```
$ ./start_ethereum.sh
```

If the command launches a Geth JavaScript console without errors, the Ethereum client is successfully initialized.

To test the Python environment, execute the following command in the current directory to run our Python script. Ensure not to terminate the Geth JavaScript console opened in the previous step:

```
$ python3 ./mpfuzz.py
```

If the script begins printing out fuzzing logs instead of returning an error, it indicates that the environment is correctly configured and ready for running the MPFUZZ prototype.

A.4 Evaluation workflow

In this section, we describe the procedures for running our artifact and reproducing the experimental results to validate the claims made in our original paper.

A.4.1 Major Claims

The experimental results obtained from running our artifact validate three major claims presented in our original paper, as listed below:

(C1): MPFUZZ discovers new asymmetric-DoS vulnerabilities. This is proven by the experiment (E1) described in § A.4.2 whose results are reported in our original paper Section 6.1.

(C2): MPFUZZ explores mempool states using symbolized transaction sequences and efficiently searches with feedback from state coverage and intermediate state promisingness. This is proven by the experiment (E1) described in § A.4.2. The experiment result illustrates how MPFUZZ finds exploits that are described in our original paper Section 5 and Appendix B.

(C3): MPFUZZ can find the first exploit, namely XT_3, on a small MUT (6 slots) in 0.03 minutes. For the medium setting, with 16 slots, MPFUZZ can find XT_3 exploit in under a minute. This is proven by the experiment (E2) described in § A.4.2 whose results are reported in our original paper Section 7.1.

A.4.2 Experiments

This subsection describe how to reproduce the experimental result that valid the Major Claims C1, C2 and C3 provided in the previous subsection. Specifically, experiment E1 validates Major Claims C1 and C2, while Major Claim C3 is validated in experiments E2.

(E1): Stateful Mempool Fuzzing by MPFUZZ [15 seconds compute-time]: In this experiment, we run MPFUZZ against a small-size Ethereum mempool to find short exploits on this mempool under test. The experimental results report all the exploits discovered by MPFUZZ and illustrate the process that MPFUZZ searches the transaction space and identifies exploits.

Preparation: First, access our artifact by following the steps described in Section A.2.2. Then, set up the environment by configuring and installing the dependencies as described in § A.3 and § A.3.1.

Execution: First, start the Ethereum client by running the following command in the current directory.

```
# In terminal 1
```

```
$ ./start_ethereum.sh
```

After the command launches a Geth JavaScript console, running the following command in a new terminal to run the MPFUZZ.

```
# In terminal 2
$ python3 ./mpfuzz.py
```

Results: After the Python script terminates, a PDF file is generated that reports the experimental results. In the PDF, nodes highlighted in yellow represent the end states of exploits, indicating that these states trigger the bug oracle. The path from the root node to the leaf node that triggers the bug oracle illustrates the exploit transaction sequence. The concrete exploit transaction sequence is also output in the Python console, allowing users to reproduce the attacks. Additionally, the tree structure in the PDF file shows the process that MPFUZZ explores the search space.

(E2): MPFUZZ performance evaluation of detecting the first exploit [1 minute compute-time]: In this experiment, we run MPFUZZ against a small-setting MUT (6 slots) and a medium-setting MUT (16 slots) to find the XT_3 exploit for evaluating the performance. The experimental results report the time used by MPFUZZ to find the XT_3 exploit and the concrete exploit transaction sequence.

Preparation: First, access our artifact by following the steps described in Section A.2.2. Then, set up the environment by configuring and installing the dependencies as described in § A.3 and § A.3.1.

Execution E2a: In experiment E2a, we evaluate the performance of MPFUZZ for finding exploit against 6-slot mempool. First, start the Ethereum client by running the following command in terminal 1.

```
# In terminal 1
$./start_ethereum_e2a.sh
```

After the command launches a Geth JavaScript console, running the following command in terminal 2 to run the MPFUZZ.

```
# In terminal 2
$python3 mpfuzz_e2a.py
```

Execution E2b: In experiment E2b, we evaluate the performance of MPFUZZ for finding exploit against 16-slot mempool. First, start the Ethereum client by running the following command in terminal 1.

```
# In terminal 1
$./start_ethereum_e2b.sh
```

After the command launches a Geth JavaScript console, running the following command in terminal 2 to run the MPFUZZ.

```
# In terminal 2
$python3 mpfuzz_e2b.py
```

Results: After the Python script terminates, the concrete exploit transaction sequence of the XT3 exploit found is output in the Python console as well as the time used to find it. MPFUZZ can find the XT_3 exploit against a 6-slot MUT in 0.03 minutes and a 16-slot MUT in under a minute.

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/usenixsec2024/.