ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

USENIX'23 Artifact Appendix: CarpetFuzz: Automatic Program Option Constraint Extraction from Documentation for Fuzzing

Dawei Wang[1,2], Ying Li[1,2], Zhiyu Zhang[1,2], and Kai Chen[1,2,3*]

[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Beijing Academy of Artificial Intelligence, China
{wangdawei,liying1998,zhangzhiyu1999,chenkai}@iie.ac.cn

A Artifact Appendix

A.1 Abstract

CarpetFuzz is an NLP-based fuzzing assistance tool specifically designed for extracting constraint relationships between command-line options from documents. Our evaluation of CarpetFuzz involved a comprehensive analysis comprising an end-to-end experiment, a comparative experiment, and four submodule experiments. To facilitate the setup process, we provide a Dockerfile, which helps mitigate potential issues with environment configuration. Additionally, we offer a collection of scripts that automate experiment reproduction and effectively showcase the results obtained.

Given the nature of fuzzing-related work, reproducing the experiments conducted with CarpetFuzz necessitates a substantial amount of computational resources. Replicating all the experiments outlined in the paper requires a total of 33,600 CPU hours (across 5 repetitions). Simplifying the process would still require a minimum of 15,840 CPU hours. Consequently, we recommend employing a server with at least 32 cores to carry out these experiments, which would approximately take around 20.6 days. It's worth noting that having a higher number of cores would further enhance the efficiency of the experiments.

A.2 Description & Requirements

Our paper describes a novel technique for identifying and extracting constraints among program options from the documentation. Our artifact is a prototype of our technique named CarpetFuzz, which contains the models, fuzzers, run scripts, and documentation. We also provide a comprehensive collection of samples, run scripts, and documentation to replicate the experiments outlined in our paper with ease.

[*]Corresponding author.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact is provided as a GitHub repository: `https://github.com/waugustus/CarpetFuzz/commit/50f09eb94d33abbfe3e18184988a0c3a8f0f5612`.

A.2.3 Hardware dependencies

As a fuzzing-related work, reproducing the experiments necessitates a significant allocation of computational resources, ranging from 15,840 to 33,600 CPU hours. To ensure completion within the review process timeframe, we recommend utilizing a server with a minimum of 32 cores, which would require approximately 20.6 days. For enhanced fault tolerance and expediency, we strongly advise opting for a server with a higher core count. In terms of hard disk capacity, our Docker image occupies around 20GB of disk space, so a disk capacity of 50GB is more than sufficient.

For the sole purpose of running CarpetFuzz, we believe that mainstream computers available on the market are sufficient to meet the requirements, such as computers with a 1-core CPU, 8GB RAM, and a 128GB hard drive.

A.2.4 Software dependencies

All software dependencies have been successfully resolved within our provided Dockerfile which is based on Ubuntu 20.04. Therefore, any system capable of running this image is suitable for the task.

A.2.5 Benchmarks

Our benchmark includes a total of 50 executable programs, with 20 sourced from our real-world program dataset and 30 obtained from the POWER dataset. All of these programs can be readily acquired from the internet. The process of obtaining and compiling each program has been thoroughly documented in our Dockerfile, facilitating automated building using the "docker build" command.

A.3 Set-up

Clone the artifact repository:

```
$ git clone --recursive https://github.com/
waugustus/CarpetFuzz; cd CarpetFuzz
```

A.3.1 Installation

For easy installation, we offer a ready-to-use Docker image for download,

```
$ sudo docker pull 4ugustus/carpetfuzz
```

or you can compile the image yourself using the Dockerfile we provide.

```
$ sudo docker build -t
4ugustus/carpetfuzz:latest .
```

Then you can create the container based on the image,

```
$ sudo docker run -it --name "carpetfuzz"
4ugustus/carpetfuzz:latest bash
```

A.3.2 Basic Test

We take the program *"tiffcp"* as an example (in the contatiner),

1. Use CarpetFuzz to analyze the relationships from the manpage file:

   ```
   $ cd /root/programs/libtiff
   ```

   ```
   $ python3 ${CarpetFuzz}/scripts/find_
   relationship.py --file $PWD/build_
   carpetfuzz/share/man/man1/tiffcp.1
   ```

2. Use pict to generate 6-wise combinations:

   ```
   $ python3 ${CarpetFuzz}/scripts/generate_
   combination.py --relation ${CarpetFuzz}/
   output/relation/relation_tiffcp.json
   ```

3. Rank each combination with its dry-run coverage:

   ```
   $ python3 ${CarpetFuzz}/scripts/rank_
   combination.py --combination ${CarpetFuzz}/
   output/combination/combination_tiffcp.txt
   --dict ${CarpetFuzz}/tests/dict/dict.json
   --bindir $PWD/build_carpetfuzz/bin
   --seeddir input
   ```

4. Fuzz with the ranked stubs:

   ```
   $ ${CarpetFuzz}/fuzzer/afl-fuzz -i
   input/ -o output/ -K ${CarpetFuzz}/
   output/stubs/ranked_stubs_tiffcp.txt --
   $PWD/build_carpetfuzz/bin/tiffcp @@
   ```

A.4 Evaluation workflow

In our paper, we evaluated CarpetFuzz through a total of six experiments, including an end-to-end experiment, a comparative experiment, and four submodule experiments, which collectively required 33,600 CPU hours. Please note that due to the authors of POWER **declining** our request to use their tool during the review process, the comparative experiment was deemed unnecessary (RQ5), resulting in a reduction of 7,200 CPU hours. Furthermore, all experiments in the paper were repeated five times. However, for simplification purposes, we consider three repetitions to be acceptable, resulting in a reduction of 10,560 CPU hours. As a result, the minimum required time for the experiments is 15,840 CPU hours.

If you desire to replicate the experiments in the paper comprehensively (excluding running POWER), you can execute "`$ ./run_fuzzing.sh`" without any options in A.4.1. This will trigger CarpetFuzz to perform fuzzing on the entire benchmark and repeat the process five times, thus amounting to a total runtime of 30,000 CPU hours.

A.4.1 Preprocessing

1. Build docker image for experiments *[1 human-minute + 6 compute-hours + 20GB disk]*:

   ```
   $ git clone https://github.com/waugustus/
   CarpetFuzz-experiments
   ```

   ```
   $ sudo docker build -t
   carpetfuzz-experiment:latest .
   ```

   ```
   $ sudo docker run -d --name
   "carpetfuzz-experiment"
   carpetfuzz-experiment:latest tail -f
   /dev/null
   ```

   ```
   $ sudo docker exec -it
   carpetfuzz-experiment bash
   ```

2. Start all fuzzing instances *[1 human-minute + 15,840 CPU-hours + 10GB disk]*:

   ```
   $ screen -dmS fuzzing bash -c
   "./run_fuzzing.sh -r 3 2>&1 |tee
   fuzzing.log"
   ```

3. Analyze the documents from the compiled programs and generate the results of relationship identification and extraction *[1 human-minute + 10 computer-minutes + 1GB disk]*:

   ```
   $ screen -dmS analyze python3
   analyze_manpages.py 2>&1 | tee analyze.log
   ```

A.4.2 Major Claims

(C1): Compared to aflfast, mopt, afl++, CarpetFuzz can help afl find more uncovered edges. This is proven by the experiment (E1) described in Section 5.1 whose results are illustrated in Table 1.

(C2): For explicitly declared relationships, CarpetFuzz achieves an accuracy of 92.90% on the validation set and 98.80% on the documentation of the 20 programs. For implicitly declared relationships, CarpetFuzz achieves an precision of 95.87% and a recall of 90.09%. This is proven by the experiments (E2) described in Section 5.2.

(C3): The precision and recall of conflict were 95.83% and 89.40%, and those of dependency were 100% and 81.82%. The precision and recall of all relationships were 96.10% and 88.85%. This is proven by the experiments (E3) described in Section 5.3.

(C4): With our prioritization technique, CarpetFuzz found more edges on each program that other fuzzers could not discover. This is proven by the experiments (E4) described in Section 5.4 whose results are illustrated in Table 2.

(C5): CarpetFuzz can discover real-world vulnerabilities. This is proven by the experiments (E4) described in Section 5.6 whose results are illustrated in Table 4.

A.4.3 Experiments

(E1): *[5 human-minutes + 1 compute-hour]: This experiment will measure the edge coverage for all Fuzzing instances and present the results in the format shown in Table 1.*

How to: First, run `get_stubs.py` in `experiments/RQ1/scripts` to collect all testcases. Second, run `afl-showmap.py` to obtain the coverage data. Third, run `analyze_results.py` to present the results.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ1/scripts
$ python3 get_stubs.py
$ python3 afl-showmap.py
$ python3 analyze_results.py
```

Results: The ouput should match Table 1 of the paper.

(E2): *[5 human-minutes + 5 compute-minutes]: This experiment will measure the relationship identification performance of CarpetFuzz on the validation set and the documentation of the 20 programs.*

How to: Run `analyze_results.py` in `experiments/RQ2/scripts` to obtain the results.

Preparation: The preprocessing step in A.4.1 is required to obtain prediction results.

Execution: Execute the following commands:

```
$ cd experiments/RQ2/scripts
$ python3 analyze_results.py
```

Results: The ouput should match Section 5.2 of the paper.

(E3): *[5 human-minutes + 5 compute-minutes]: This experiment will measure the relationship extraction performance of CarpetFuzz.*

How to: Run `analyze_results.py` in `experiments/RQ3/scripts` to obtain the results.

Preparation: The preprocessing step in A.4.1 is required to obtain extraction results.

Execution: Execute the following commands:

```
$ cd experiments/RQ3/scripts
$ python3 analyze_results.py
```

Results: The ouput should match Section 5.3 of the paper.

(E4): *[5 human-minutes + 1 compute-hour]: This experiment will measure the edge coverage for CarpetFuzz-random instances and present the results in the format shown in Table 2.*

How to: First, run `get_stubs.py` in `experiments/RQ4/scripts` to collect all testcases. Second, run `afl-showmap.py` to obtain the coverage data. Third, run `analyze_results.py` to present the results.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ4/scripts
$ python3 get_stubs.py
$ python3 afl-showmap.py
$ python3 analyze_results.py
```

Results: The ouput should match Table 2 of the paper.

(E5): *[5 human-minutes + 1 compute-hour]: This experiment will tally the number of crashes encountered by CarpetFuzz instances.*

How to: Run `analyze_results.py` in `experiments/RQ6/scripts` to tally the number of crashes.

Preparation: The preprocessing step in A.4.1 is required to have fuzzing results.

Execution: Execute the following commands:

```
$ cd experiments/RQ6/scripts
$ python3 analyze_results.py
```

Results: CarpetFuzz should find multiple crashes in the 20 programs.

A.5 Notes on Reusability

A.5.1 How to find the manpage file of a new program?

In our experience, manpage files are typically located in the share directory within the compilation directory, such as `/your_build_dir/share/man/man1`.

A.5.2 How to fuzz the target not recorded in dict.json?

Unfortunately, as mentioned in the paper, CarpetFuzz does not currently support the automatic selection of appropriate option values from the document. To fuzz a new program, you'll need to read the manpage and manually add the synopsis and option-value pairs in the JSON, which may not be too time-consuming.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2023/`.