# DAFL: Directed Grey-box Fuzzing guided by Data Dependency (Paper Artifact)

This is the artifact of the paper *DAFL: Directed Grey-box Fuzzing guided by Data Dependency* to appear in USENIX Security 2023.

## Instructions

1. Download the file `DAFL-artifact.tar.gz` and extract the files with the following command:

```
tar -zxvf DAFL-artifact.tar.gz
```

2. Move down to the directory `DAFL-artifact` and start referring to the rest of the sections in this document.

The rest of the contents in this document is the same as the README in `DAFL-artifact`.

## 1. Getting started

### 1.1. System requirements

To run the experiments in the paper, we used a 64-core (Intel Xeon Processor Gold 6226R, 2.90 GHz) machine with 192 GB of RAM and Ubuntu 20.04. Out of 64 cores, We utilized 40 cores with 4 GB of RAM assigned for each core.

The system requirements depend on the desired experimental throughput. For example, if you want to run 10 fuzzing sessions in parallel, we recommend using a machine with at least 16 cores and 64 GB of RAM.

You can set the number of iterations to be run in parallel and the amount of RAM to assign to each fuzzing session by modifying the `MAX_INSTANCE_NUM` and `MEM_PER_INSTANCE` variables in `scripts/common.py`. The default values are 40 and 4, respectively.

Additionally, we assume that the following environment settings are met.

- Ubuntu 20.04
- Docker
- python 3.8+
- pip3

For the Python dependencies required to run the experiment scripts, run

```
yes | pip3 install -r requirements.txt
```

### 1.2. System configuration

To run AFL-based fuzzers, you should first fix the core dump name pattern.

```
$ echo core | sudo tee /proc/sys/kernel/core_pattern
```

If your system has `/sys/devices/system/cpu/cpu*/cpufreq` directory, AFL may also complain about the CPU frequency scaling configuration. Check the current configuration and remember it if you want to restore it later. Then, set it to `performance`, as requested by AFL.

```
$ cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
powersave
powersave
powersave
powersave
$ echo performance | sudo tee
/sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

### 1.3. Preparing the Docker image

Our artifact is composed of two parts: the Docker image and the framework to build and utilize it. The Docker image contains all the necessary tools and dependencies to run the fuzzing experiments. The framework, which holds this README file, is used to build the Docker image and orchestrate the fuzzing experiments.

**Recommended** You can pull the pre-built Docker image from Dockerhub.

To do so, run

```
$ docker pull prosyslab/dafl-artifact
```

The image is big (around 25 GB) and it may take a while to download.

**DIY** If you want to build the docker image yourself, run

```
docker build -t prosyslab/dafl-artifact -f Dockerfile .
```

However, we do not recommend this because it will take an extremely long time (up to 3~4 days) to build. Nonetheless, we provide the Docker file and the relevant scripts to show how the Docker image was built.

## 2. **Directory structure**

### 2.1. Local framework structure

```
├─ README.md                    <- The top-level README (this file)
│  │
├─ docker-setup                 <- All scripts and data required to build
the Docker image
│  │
│  ├─ benchmark-project         <- Directory to build each benchmark
project-wise
│  │  ├─ binutils-2.26
│  │  │  ├─ poc                 <- POC files for each target in binutils-
2.26
│  │  │  └─ build.sh            <- The build script to build binutils-2.26
│  │  ├─ ...
│  │  │
│  │  └─ build-target.sh        <- A wrapper script to build each
benchmark project
│  │
│  ├─ target                    <- Target program locations for each
target
│  │  ├─ line                   <- Target lines
│  │  └─ stack-trace            <- Target stack trace (for AFLGo)
│  │
│  ├─ tool-script               <- Directory for fuzzing scripts
│  │  └─ run_*.sh               <- The fuzzing scripts to run each fuzzing
tool
│  │
│  ├─ Beacon-binaries           <- Binaries extracted from the provided
Docker image
│  │                              (yguoaz/beacon: Docker SHA256 hash
a09c8cb)
│  │
│  ├─ windranger.tar.gz         <- Implementation of WindRanger extracted
from the provided Docker image
│  │                              (ardu/windranger: Docker SHA256 hash
8614ceb)
│  │
│  └─ setup_*.sh                <- The setup scripts to setup each fuzzing
tool
│  └─ build_bench_*.sh          <- The build scripts to build the target
programs
│
├─ DAFL                         <- Implementation of DAFL, the fuzzer
├─ DAFL_energy                  <- Variant of DAFL that uses only utilizes
energy scheduling
│
├─ sparrow                      <- Implementation of Sparrow, the static
analyzer
│
├─ scripts                      <- Directory for scripts
│
├─ output                       <- Directory where outputs are stored
```

```
│
├─ scripts                        <- Directory for scripts
│
├─ Dockerfile                     <- Docker file used to build the Docker
image
│
├─ sa_overhead.csv                <- Static analysis overheads for each tool
│
└─ requirements.txt               <- Python requirements for the scripts
```

### 2.2. Docker directory structure

```
├─ benchmark                      <- Directory for benchmark data
│  │
│  ├─ bin                         <- Target binaries built for each fuzzing
tool
│  │  ├─ AFL
│  │  ├─ ...
│  │
│  ├─ target                      <- Target program locations for each
target
│  │  ├─ line                     <- Target lines
│  │  └─ stack-trace              <- Target stack trace (for AFLGo)
│  │
│  │
│  ├─ poc                         <- Proof of concept inputs for each target
│  │
│  ├─ seed                        <- Seed inputs for each target
│  │
│  ├─ smake-out                   <- Input files for Sparrow, the static
analyzer
│  │
│  ├─ DAFL-input                  <- Input files for DAFL
│  │  ├─ dfg                      <- Data flow graph
│  │  └─ inst-targ                <- Instrumentation targets (list of
functions)
│  ├─ DAFL-input-naive            <- Input files for DAFL, but with naive
slicing
│  │
│  └─ build_bench_*.sh            <- The build scripts to build the target
programs
│                                    for each fuzzing tool
│
├─ fuzzer                         <- Directory for fuzzing tools
│  ├─ AFL                         <- The initial rule used for
│  ├─ ...
│  └─ setup_*.sh                  <- The setup scripts to setup each fuzzing
tool
│
├─ tool-script                    <- Directory for fuzzing scripts
│  └─ run_*.sh                    <- The fuzzing scripts to run each fuzzing
```

```
tool
|
└─ sparrow                        <- Implementation of Sparrow, the static
analyzer
```

# 3. **Reproducing the results in the paper**

## 3.1. Running the experiment on specific targets

To run the experiment on specific targets, you can run

```
$ python3 ./scripts/reproduce.py run [target] [timelimit] [iteration] [tool
list]
```

For example, you can run the experiment on CVE `2018-11496` in `lrzip-ed51e14` for 60 seconds and 40 iterations with the tool `AFL`, `AFLGo`, `WindRanger`, `Beacon`, and `DAFL` by the following command.

```
$ python3 ./scripts/reproduce.py run lrzip-ed51e14-2018-11496 60 40 "AFL
AFLGo WindRanger Beacon DAFL"
```

The result will be parsed and summarized in a CSV file, `lrzip-ed51e14-2018-11496.csv`, under `output/lrzip-ed51e14-2018-11496-60sec-40iters`.

For the available choices of targets, refer to the `FUZZ_TARGETS` in `scripts/benchmark.py`.

## 3.2. Running the experiments in each table and figure

To reproduce the results in each table and figure, you can use the script `scripts/reproduce.py` as the following.

```
$ python3 ./scripts/reproduce.py run [table/figure name] [timelimit in
seconds] [iterations]
```

First, the corresponding fuzzing experiment will be run.
Then the results are saved under `output/[table/figure]-[timelimit]sec-[iteration]iters`.
Finally, the result will be parsed and summarized in a CSV file, `[table/figure].csv`, under the corresponding output directory.

For example, by the following command,

```
$ python3 ./scripts/reproduce.py run tbl2 86400 40
```

6 fuzzers, AFL, AFLGo, WindRanger, Beacon, and DAFL (with and without ASAN to compare with Beacon) will be run on all targets for 24 hours and 40 iterations to reproduce the results in Table 2. The results will be stored under `output/tbl2-86400sec-40iters` and parsed to `output/tbl2-86400sec-40iters/tbl2.csv`.

FYI, you can choose from the following table/figure names.

- `tbl2, fig7, fig8, fig9`

### 3.3. Running the scaled down version of 3.2

Reproducing the experiments in our paper at a full scale will take a very long time with limited resources. For example, we ran a 24-hour fuzzing session with 6 fuzzers on 41 targets, each repeated 40 times for the main experiment described in Section 4.2 of our paper. If run on a single machine capable of running 40 fuzzing sessions in parallel, this experiment takes 246 days of fuzzing time. Thus, we provide a scaled-down version of the experiment that can be run in a reasonable amount of time. Under the assumption of running the experiment on a machine that is capable to run 40 fuzzing sessions in parallel, each scaled-down version of the experiment can be run in 1~4 days.

This scaled down version comprises the following four targets:

- swftophp-4.8-2018-11225
- swftophp-4.8-2019-12982
- xmllint-2017-9048
- cjpeg-1.5.90-2018-14498

These targets are chosen because they clearly demonstrate the effectiveness of DAFL, while other fuzzers have hard time reproducing them.

The iterations in the scaled-down version of the experiment are also reduced to 10. Furthermore, for the experiment in Table 2, we only run the experiment where ASAN options are enabled, thus only comparing AFL, AFLGo, WindRanger, and DAFL.

To run the scaled-down version of the experiment, run the experiment with the `-scaled` attached to the target argument. For example, to run the scaled-down version of the experiment in Table 2, run

```
$ python3 ./scripts/reproduce.py run tbl2-scaled 86400 10
```

This will run four fuzzers, AFL, AFLGo, WindRanger, DAFL on aforementioned four targets, each repeated 10 times for 24 hours.

Note that running AFL and DAFL is common to all experiments (Table 2, Figure 7, Figure 8, Figure 9). Once you have run the experiment for Table 2, the experiment script automatically reuses the results from the

experiment of Table 2 for the following experiments to further reduce the fuzzing time. Just make sure that the results are under the directory `output/tbl2-scaled-86400sec-10iters`.

### 3.4. Parsing the results

If you have already run the fuzzing sessions and only want to parse the results, you can run

```
$ python3 ./scripts/reproduce.py parse [figure/table/target] [timelimit]
[iteration]
```

If the corresponding output directory exists in the form of `[table/figure/target]-[timelimit]sec-[iteration]iters`, the existing fuzzing result will be parsed and summarized in a CSV file, `[figure/table/target].csv`, under the corresponding output directory

## 4. The results of the experiments in the paper

You can retrieve the results of the experiments in the paper from here. Download the file `DAFL_experiments.tar.gz`, extract the files, and move them to the appropriate location with the following command. Note that the size of the file after extraction is about 48GB, so be sure to have enough storage space

Run

```
wget or download the archived file
tar -xvf [file name]
mv [directory name] output/origin
```

Then you can parse the results by running the following command to get the same results as in the paper.

```
$ python3 ./scripts/reproduce.py parse [origin-(figure/table)] 86400 40
```

Note that you must notate the desired target with the prefix `origin-`.

For example, if you want to get the original results used for Figure 6, run

```
$ python3 ./scripts/reproduce.py parse origin-fig6 86400
```

The CSV file will be stored in the `output/origin` directory.