APPENDIX A
ARTIFACT APPENDIX

In this artifact, we provide a detailed guidance to deploy Sysyphuzz and reproduce the experiments presented in the paper.

A. *Description and Requirements*

This artifact provides the necessary materials to reproduce the results for the following research questions:

- **RQ1:** Can SYSYPHUZZ increase the hit counts of low-frequency kernel code? The corresponding results are presented in Section VI, Figure 7 of the manuscript.
- **RQ2:** Can SYSYPHUZZ outperform SYZKALLERin terms of coverage and bug discovery? The results are reported in Section VI, Figures 8–11 and Table II.
- **RQ3:** Is the bug discovery performance of SYSYPHUZZ correlated with low-frequency code regions? The supporting evidence is presented in Section VI, Figure 12.

1) How to access: The code is available in zendo https://tinyurl.com/4u3mzay9

2) Hardware dependencies: This artifact requires at least 16 CPU cores, 64 GB of RAM and a 128 GB disk.

3) Software dependencies: This artifact requires a Linux environment with all dependencies configured as required by Syzkaller.

B. *Artifact Installation & Configuration*

Run the *deploy.sh* script with sudo privileges. This script creates a new user *fuzz* and sets up the environment. You can modify the default password in the shell file if needed. Once executed, Sysyphuzz is ready for use.

C. *Experiment Workflow*

Our experimental workflow consists of two steps. First, we run SYSYPHUZZ and SYZKALLER for over three days to collect the necessary data. Due to hardware constraints, the server supports running only one fuzzer at a time. As a result, users must manually alternate between executing SYSYPHUZZ and SYZKALLER during the experiment.

```
# Make sure you are the user "fuzz"
# and in the sysyphuzz directory
su fuzz
cd ~/code/sysyphuzz/
# Start Sysyphuzz
sudo bin/syz-manager \
-config sysyphuzz.cfg 2>&1 \
| tee ./workdir_sysy/"$(date +"%Y_%m_%d").log"
# Useing "ctrl + c" to stop.
# Change to syzkaller.cfg and workdir_syzk
```

Listing 1. Running Bash Cmds

After the first step, two types of logs will be generated: *fuzzing logs* and *hit count logs*. The former are stored in the `workdir_sysy` or `workdir_syzk` directories, while the latter are saved in the corresponding `workdir_sysy_bb` or `workdir_syzk_bb` directories.

All directory paths can be customized in the respective `*.cfg` configuration files. The above experiment should be repeated to ensure that four rounds of results are collected for both SYSYPHUZZ and SYZKALLER.

```
# Fuzzing logs
-/home/fuzz/code/sysyphuzz/workdir_sysy #_syzk
  -YYYY-MM-DD.log
  -crashes
# Hit count logs
-/home/fuzz/code/sysyphuzz/workdir_sysy_bb #
    _syzk_bb
  -YYYYMMDD_hh
    -HitBBPerc_hhmm.json
    -UnderCoveredBBs_hhmm.json
```

Listing 2. Log Structure

Run the following analysis scripts to reproduce the evaluation results and support the main claims of the paper.

D. *Major Claims*

- (C1): SYSYPHUZZ effectively balances the kernel code coverage by improving hit counts and reducing the survival rate of initial low-frequency regions. This is proven by the experiment (E1) whose results are illustrated in VI-Fig7.
- (C2): SYSYPHUZZ discovers more unique bugs missed by SYZKALLER and slightly outperforms SYZKALLER in code coverage. This is proven by the experiment (E2) whose results are illustrated in VI-Fig8 to Fig11.
- (C3): SYSYPHUZZ's bugs discovered by SYSYPHUZZ are largely related to underexplored low-frequency regions. This is proven by the experiment (E2) whose results are illustrated in VI-Fig12.

E. *Evaluation*

1) Experiment (E1): [Low-frequency areas analysis] [60 human-minutes + 1 compute-hour]: For the six observation time points, we extract the hit counts of the low-frequency regions and compute the survival rate of the initially identified low-frequency regions across subsequent time points.

[How to] In each experimental trial, collect hit count logs at six time points. Run two scripts: one to extract low-frequency regions at each point, and another to compute the survival rate of the regions identified at the first time point.

[Preparation] For each trial, collect hit count logs at six time points: t0, t0+6h, t0+12h, t0+24h, t0+48h, and t0+72h. The reference point t0 marks the completion of corpus replay and the start of fuzzing, which is the creation time of the first hit count log.

```
# Example of the Hit count log create time.
-/home/fuzz/code/sysyphuzz/workdir_sysy_bb
  -YYYYMMDD_hh
    -UnderCoveredBBs_hhmm.json
```

Listing 3. Get Hit Count Logs

The timestamp in the format YYMMDD_hhmm corresponds to the creation time of each UnderCoveredBBs JSON file. We identify the first available file as the t0 time point and manually compute the subsequent five time points. If a required file is missing, we substitute it with the nearest available file from an earlier time. Each of the selected files should be copied to a new directory and renamed as t0.json, t0+6h.json, etc. This process is repeated for all trials in the experiment.

[Execution] For each trial of the experiments, get the low-frequency areas in six time points by using script *get_low_area.py*. The input files are the six json files created in the preparation step. There will be six json files in the OUTPUT_DIR, namely filtered_t0.json_1, filtered_t0+6h.json_2, etc.

```
# --per is 5 in our experiments.
python3 get_low_area.py -h
usage:
--input_files INPUT_FILES [INPUT_FILES ...] \
--per PER --output_dir OUTPUT_DIR
```

Listing 4. Get Low-Frequency Areas

Repeat the above step for the rest trials of the experiments. For four trials of the experiments, there will be four filtered json files for each time point, e.g., trial1_filtered_t0.json_1, trial2_filtered_t0.json_1, etc. Using the script *get_average_hit.py* to get the average hit counts of the low-frequency areas at t0, repeat this step for the rest time points.

```
python3 get_average_hit.py -h
usage:
--input_files INPUT_FILES ...  \
--output_file OUTPUT_FILE
```

Listing 5. Get Average Hit Counts

Then, using *get_consist.py* to compute the survival rate of the low-frequency areas identified at t0, the script processes each trial by reading the BBs from trial*_filtered_t0.json and evaluating their presence in the subsequent time points logs, e.g., trial*_filtered_t0+6h.json, trial*_filtered_t0+12h.json, etc.

```
python3 get_consist.py -h
usage:
--input_files INPUT_FILES ...  \
--output_file OUTPUT_FILE
```

Listing 6. Get Survival Rate

Repeat the above steps for the remaining trials to generate four survival rate JSON files in total. Then, use the script *get_average_survival.py* to compute the average survival rate of the initial low-frequency areas.

```
python3 get_average_survival.py -h
usage:
--input_files INPUT_FILES [INPUT_FILES ...]  \
--output_file OUTPUT_FILE
```

Listing 7. Get Average Survival Rate

[Results] The average hit counts and survival rates will be used in the VI-Fig7.

2) Experiment (E2): [Coverage and bug analysis] [hundred human-hours + tens compute-hours]: Based on the fuzzing logs under workdir, running the script *plog_coverage.py* to draw the coverage, bug performance, and bug overlap figures.

```
# MAX_HOURS is hours after corpus replay,
# e.g., 72 means t0+72.
python3 plog_coverage.py -h
usage:
--boost_logs BOOST_LOGS ... \
--coverage_logs COVERAGE_LOGS ... \
--output OUTPUT [--max_hours MAX_HOURS]
```

Listing 8. Coverage Bash Cmds

[Results] The output figure will be used in the VI-Fig9.

Running the script *plog_coverage_batch_totoalcrash_1.py* to get overall bug performance, and use the script *classify.py* to classify the crashes based on different bug types.

```
plog_coverage_batch_totoalcrash_1.py -h
usage:
--boost_logs BOOST_LOGS  \
--coverage_logs COVERAGE_LOGS \
--output OUTPUT \
--crash_report CRASH_REPORT \
[--max_hours MAX_HOURS]
#  Classify Bugs.
python3 classify.py -h
usage:
--input_json INPUT_JSON \ # CRASH_REPORT
--output_dir OUTPUT_DIR
```

Listing 9. Bug Analysis Bash Cmds

[Results] The data will be used to draw the VI-Fig8,10,11.

For each unique bug, extract the crash stack trace from the crash log and the first trigger time from the fuzzing log. Identify and merge all low-frequency areas identified by SYSYPHUZZ prior to the bug trigger time. Finally, compute the overlap between these merged low-frequency areas and the bug's crash stack trace.

```
python3 extract_crash_trace_segment.py -h
usage:
-i INPUT [-o OUTPUT]
# Getting the merged low-frequency areas.
python3 low_bb_bef_crash_time.py -h
usage:
-syzlog SYZLOG \ # Sysyphuzz fuzzing log
-bblog BBLOG \ # related hit count log
    directory
-vmlinux VMLINUX \ # vmlinux binary
-time_stamp TIME_STAMP \ # first trigger time
-o O
# Find the overlap.
python3 overlap_analysis.py -h
usage:
-crash_csv CRASH_CSV  -lf_csv LF_CSV [-o O]
```

Listing 10. Bug Overlap Bash Cmds

[Results] The data will be used in the VI-Fig12.

F. Notes

This artifact is under major revision. The next version will include additional comparisons with SyzGPT and extended analysis.