PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns

Artifact Overview

1 Introduction

This artifact aims to reproduce the results of PAFL in our paper "PAFL: Enhancing Fault Localizers by Leveraging Project-Specific Fault Patterns" submitted to OOPSLA 2025. By following this manual, you can replicate the results of PAFL as shown in Tables 2, 3, 4, 6, and 7 of the paper. Additionally, we provide instructions on how to reuse this artifact for further applications.

2 Hardware Dependencies

By default, this artifact is configured to reproduce the results of the *cpp_peglib* project, requiring 2GB of disk space. This default setup is chosen due to the substantial storage requirements for reproducing all results. To reproduce the results for all projects, 200GB of disk space is necessary. The complete datasets can be accessed via the following links: `https://doi.org/10.5281/zenodo.14583938` and `https://figshare.com/s/8d82745d78ade3bbab5d`. To enable the artifact to reproduce results for all projects, extract the files and place them in the "`data/source`" and "`data/coverage`" directories.

3 Getting Started Guide

3.1 Setup

We provide a Docker image that contains all the dependencies. The following command builds the Docker image (this takes about 5 minutes):

```
$ ./install.sh
```

The command will start the Docker container.

3.2 Verifying Installation (Basic Testing)

To verify the installation, please run the following command:

```
$ train_and_run ochiai cpp_peglib
```

The necessary project sources and code coverage files for training and evaluation are stored in the directories, "`data/source`" and "`data/coverage`". The above command will train and run baseline FL (Ochiai) and PAFL for each version of the *cpp_peglib* project. If the artifact is successfully installed, you will see the following results:

```
...
=======================================
#Used threads: 1
Elapsed time (ochiai)       : 0.1 secs
Elapsed time (ochiai + PAFL): 3.6 secs
Average overhead: 0.4 secs
=======================================
```

The above results illustrate that the training and evaluation took 0.4 seconds on average for each version of the *cpp_peglib* project. The elapsed time reported in the above results varies depending on the experiment environment. In our work, all the experiments on PAFL are conducted on a machine with Intel Xeon Silver 4214 with 12 cores.

4 Step by Step Instructions

The following instructions will reproduce the results in Table 2, 3, 4, 6, and 7.

4.1 Reproducing Table 2

Following the instructions reproduce Table 2 of our paper.

Training and running PAFL models. To reproduce the results in Table 2 (PAFL for the SBFL baselines), start by training PAFL using the following command:

```
$ train_and_run <baseline> <project> -t <numthreads>
```

where `baseline` can be one of the following SBFL baselines:

```
ochiai, dstar, barinel
```

and `project` can be one of the following projects[1].

```
cpp_peglib, cppcheck, exiv2, libchewing, libxml2, proj,
openssl, yaml_cpp, thefuck, fastapi, spacy, youtube-dl
```

In the above commane, `numthreads` determines the number of threads (default = 1) for training and evaluating the models. For example, if you type `$ train_and_run ochiai cpp_peglib -t 8`, the command will train the model and produce the output of Ochiai and PAFL for each version of the *cpp_peglib* project using eight threads. The trained model will be stored in the "`/opt/PAFL/profile`" directory. If the training and evaluation are successfully finished, the command will produce the following results:

```
...
======================================
#Used threads: 8
Elapsed time (ochiai)       : 0.0 secs
Elapsed time (ochiai + PAFL): 1.4 secs
Average overhead: 0.1 secs
======================================
```

The above results illustrate that the training and running use eight threads and took 0.1 seconds on average for each version of the *cpp_peglib* project.

Reproducing the results in Table 2. After the training, type the following command to reproduce the results in Table 2:

```
evaluate <baseline> <project>
```

Then, the command will reproduce *FR* of each version, and *MFR*, *MAR*, *Top-1*, *Top-5*, and *Top-10* for the project in Table 2. For example, if you type `$ evaluate ochiai cpp_peglib`, the command will produce the following results:

[1] As noted in Section 2, the datasets projects (other than *cpp_peglib*) must first be downloaded and placed in the data/source and data/coverage directories before running the above command.

```
...
cpp_peglib#10
================================
    |   ochiai    |    PAFL
--------------------------------
 FR |    4  /1775 |    2  /1775
 AR |  247.3/1775 |  240.9/1775
================================

Total
========================================
        |    ochiai     |     PAFL
----------------------------------------
 MFR    |  151.0/1426.6 |  138.8/1426.6
 MAR    |  208.6/1426.6 |  195.4/1426.6
 Top-1  |          1    |          1
 Top-5  |          3    |          3
 Top-10 |          3    |          3
========================================
```

The above results illustrate that *FR* and *AR* of PAFL for version *cpp_peglib*#10 are 2 and 240.9, respectively. The total results (i.e., `Total`) reproduced the cell corresponds to row "*cpp_peglib*" and column "Ochiai" and "PAFL" in Table 2. For example, *MFR*, *MAR*, and *Top-10* of PAFL for the *cpp_peglib* project are 138.8, 195.4, and 3, respectively. Other cells in Table 2 can be reproduced by changing the project and baseline in the command.

4.2 Reproducing Table 3

Since the baselines in Table 3 are deep learning-based techniques with inherent stochasticity, the following instructions may yield results that differ slightly from those reported in the paper. For reproducing the results of PAFL for the DLFL baselines in Table 3, run the following command:

```
$ evalDLFL <baseline> <project> -t <numthreads>
```

`baseline` can be one of the following DLFL baselines:

`CNN, RNN, MLP`

`project` can be one of the following projects for `CNN` and `RNN`:

`cpp_peglib, libchewing, thefuck, fastapi, spacy`

`project` can be one of the following projects for `MLP`:

`cpp_peglib, cppcheck, exiv2, libchewing, libxml2, proj,`
`yaml_cpp, thefuck, fastapi, spacy, youtube-dl`

For example, if you type `$ evalDLFL CNN cpp_peglib -t 8`, the command will produce the following results:

```
...
Average metrics for 5 iterations:
========================================
        |      CNN      |     PAFL
----------------------------------------
 MFR    |  328.6/1426.6 |  300.3/1426.6
 MAR    |  399.3/1426.6 |  385.7/1426.6
 Top-1  |           0.0 |           0.0
 Top-5  |           0.0 |           0.0
 Top-10 |           0.0 |           0.0
========================================
```

The above execution attempted to reproduce the results corresponding to the row "cpppeglib" and columns "`CNN`" and "PAFL" in Table 3, but the results slightly differ from those in the paper due to the stochastic nature of the DL-based approaches.

4.3 Reproducing Table 6

The instructions below reproduce the results in Table 6.

Training and running PAFL models. To fully reproduce the results (specifically the rows labeled "together") in Table 6, start by training PAFL with AENEAS using the following command:

`$ train_and_run <baseline> <project> -t <numthreads>`

To reproduce Table 6, `baseline` can be one of the following AENEAS combined with SBFL:

`aeneas-ochiai, aeneas-dstar, aeneas-barinel`

`project` can be one of the following projects:

`cpp_peglib, cppcheck, libchewing, libxml2,`
`yaml_cpp, thefuck, fastapi, spacy, youtube-dl`

If the training and running are successful, the command (e.g., `$ train_and_run aeneas-ochiai cpp_peglib -t 8`) will produce the following results:

```
...
======================================
#Used threads: 8
Elapsed time (ochiai)       : 0.0 secs
Elapsed time (ochiai + PAFL): 1.4 secs
Average overhead: 0.1 secs
======================================
```

Reproducing the results in Table 6. To reproduce the results, run the following command:

`evaluate <baseline> <project>`

Then, the command will reproduce *MFR*, *MAR*, *Top-1*, *Top-5*, and *Top-10* for the project in Table 6. For example, if you type `$ evaluate aeneas-ochiai cpp_peglib`, the command will produce the following results:

```
...
Total
========================================
        | aeneas-ochiai |     PAFL
----------------------------------------
 MFR    |  125.0/1426.6 |  118.9/1426.6
 MAR    |  165.3/1426.6 |  159.2/1426.6
 Top-1  |        0      |        0
 Top-5  |        2      |        2
 Top-10 |        2      |        2
========================================
```

Note that the above results only show the performance of AENEAS and PAFL only for the *cpp_peglib* project. To reproduce the cells in Table 6, where each cell presents the total performance across all the project, you need to integrate the results of all the other projects into a single result. To reproduce the cells in Table 6, where each cell represents the overall performance across all projects, you need to integrate the results from all the projects into a single aggregated result.

4.4 Reproducing Table 4

The instructions below reproduce the results in Table 4 (robustness of PAFL). Before proceeding, ensure that you have fully reproduced the results in Tables 2 and 3 by following the instructions in Section 4.1 and Section 4.2.

Reproducing the results in Table 4. To reproduce Table 4, run the following command:

```
robustness <baseline> <project>
```

Then, the command will reproduce the robustness of PAFL for the project. For example, if you type `$ robustness ochiai cpp_peglib`, the command will produce the following results:

```
Baseline FL: ochiai
Target project: cpp_peglib
--------------------------
Acheiving equal or better FR in 10 out of 10 (100%) versions
```

The above results illustrate that 100% of versions of *cpp_peglib* are improved by PAFL or equal compared to Ochiai.

4.5 Reproducing Table 7

Following the instructions below reproduce the results in Table 7.

Training with various configurations. To fully reproduce the results in Table 7, train PAFL using seven different configurations with the following command:

```
$ train_and_run ochiai <project> -u <config> -t <numthreads>
```

In the above command, `config` can be one of the following seven configurations:

```
1111, 0000, 2111, 1211, 1121, 1112, 2222
```

Again, `project` can be one of the following projects:

```
cpp_peglib, cppcheck, exiv2, libchewing, libxml2, proj,
openssl, yaml_cpp, thefuck, fastapi, spacy, youtube-dl
```

Reproducing the results in Table 7. After the training, the following command will reproduce the results in Table 7:

```
evaluate ochiai <project> -u <config>
```

Then, the command will reproduce *MFR*, *MAR*, and *Top-10* for the project in Table 7. For example, if you type `$ evaluate ochiai cpp_peglib -u 2222`, the command will produce the following results:

```
...
Total
========================================
        |    ochiai     |  PAFL (2222)
----------------------------------------
 MFR    |  151.0/1426.6 |  138.8/1426.6
 MAR    |  208.6/1426.6 |  195.5/1426.6
 Top-1  |          1    |          2
 Top-5  |          3    |          3
 Top-10 |          3    |          3
========================================
```

The above results illustrate that *MFR*, *MAR*, *Top-1*, *Top-5*, and *Top-10* of the PAFL model with configuration `(2,2,2,2)` for the *cpp_peglib* project are 138.8, 195.5, 2, 3, and 3, respectively.

5 Reusability Guide

The artifact can be reused for improving other (baseline) fault localizers through the following steps:

5.1 Storing Suspiciousness Scores from Baseline Fault Localizers

To apply PAFL to a new fault localizer, you need to store the suspiciousness scores computed by the new fault localizer as follows:

```
data/coverage
|_ <method>
    |_ <project>
        |_ <version>.json
```

In the above, `method` is the name of the baseline fault localizer. `project` is the target project name. `version.json` stores suspicious score for each line of a version in the project as follows:

```
{ "lines": [
    {
        "file": <file_name>,
        "lineno": <line_number>,
        "sus": <new_score>
    },
    ...
] }
```

In the above, `file_name` is the name of the source file. `line_number` is the line number (i.e., statement) in the source file. `new_score` is the suspiciousness score of the line. For example, if you want to apply PAFL to a new FL method 'example_FL' in the *cpp_peglib* project, place the files as follows:

```
data/coverage
|_ example_FL
    |_ cpp_peglib
        |_ 1.json
        |_ 2.json
        |_ ...
        |_ 10.json
```

In the above, 1.json includes the suspicious scores of the *cpp_peglib* project for the first version computed by the 'example_FL' method. `<version>.json` have the following structure:

```
{ "lines": [
    {"file":"peglib.h", "lineno":2155, "sus": 0.4},
    {"file":"peglib.h", "lineno":2156, "sus": 0.3},
    {"file":"peglib.h", "lineno":2122, "sus": 0.2},
    ...
] }
```

The above file shows that the suspiciousness score of the 2155th line in the file 'peglib.h' is 0.4, the 2156th line is 0.3, and the 2122th line is 0.2.

5.2 Applying PAFL

To train and run PAFL with the new baseline fault localizer, run the following command:

```
$ train_and_run <method> <project> -u <config> -t <numthreads>
```

To evaluate the performance of PAFL with the new scores, run the following command:

```
$ evaluate <method> <project> -u <config>
```

For example, the following commands will produce the results of PAFL with the baseline FL method, 'example_FL', for the *cpp_peglib* project:

```
$ train_and_run example_FL cpp_peglib
```

```
$ evaluate example_FL cpp_peglib
```