A Comparative Study on the Accuracy and the Speed of Static and Dynamic Program Classifiers

Anderson Faustino da Silva[*], Jeronimo Castrillon[†], Fernando Magno Quintão Pereira[‡]

Abstract

Classifying programs based on their tasks is essential in fields such as plagiarism detection, malware analysis, and software auditing. Traditionally, two classification approaches exist: static classifiers analyze program syntax, while dynamic classifiers observe their execution. Although dynamic analysis is regarded as more precise, it is often considered impractical due to high overhead, leading the research community to largely dismiss it. In this paper, we revisit this perception by comparing static and dynamic analyses using the same classification representation: opcode histograms. We show that dynamic histograms—generated from instructions actually executed—are only marginally (4-5%) more accurate than static histograms in non-adversarial settings. However, if an adversary is allowed to obfuscate programs, the accuracy of the dynamic classifier is twice higher than the static one, due to its ability to avoid observing dead-code. Obtaining dynamic histograms with a state-of-the-art Valgrind-based tool incurs an 85x slowdown; however, once we account for the time to produce the representations for static analysis of executables, the overall slowdown reduces to 4x: a result significantly lower than previously reported in the literature.

1 Reproducibility

Implementations and datasets discussed in the paper are publicly available at https://github.com/ComputerSystemsLaboratory/Rouxinol, and https://doi.org/10.5281/zenodo.14811734.

2 Artifact Abstract

This artifact compares the accuracy and speed of static and dynamic program classifiers. The artifact consists of scripts that replicate Figures 4-8, plus the dataset.

3 Artifact Meta-Information

- **Program:** Code representation: Rouxinol[1] (LLVM histograms and X86 histograms), IR2Vec[2], Compy-Learn[3] (ProGraML), CFGGrind[4] (H-x86 and D-x86 histograms). Classification model: Random Forest (`SciKit-Learn`) and GNN[5]. Code obfuscator: `o-llvm`[6]. Environment: Anaconda[7].
- **Compilation:** clang, gcc, cmake.
- **Dataset:** The data samples are included.
- **Run-time environment:** Ubuntu.

*UEM, Brazil, afsilva@uem.br
†TU Dresden and SCADS.AI, Germany, jeronimo.castrillon@tu-dresden.de
‡UFMG, Brazil, fernando@dcc.ufmg.br

[1] https://github.com/ComputerSystemsLaboratory/Rouxinol
[2] https://github.com/IITH-Compilers/IR2Vec
[3] https://github.com/tud-ccc/compy-learn
[4] https://github.com/rimsa/CFGgrind
[5] https://github.com/tud-ccc/compy-learn
[6] https://github.com/heroims/obfuscator/tree/llvm-10.x
[7] https://www.anaconda.com/download

- **Hardware:** Any x86-64 machine with at least 64 GB of RAM memory.
- **Metrics:** Accuracy and time.
- **Output:** Figures 4-8 in PDF format.
- **How much disk space required (approx.)?:** 42 GB.
- **How much time is needed to prepare workflow (approximately)?:** 4.5 hours (using 4 jobs).
- **How much time is needed to complete experiments (approximately)?:** All the experiments take approximately 45 hours (4 workers):
  - Data generation takes $\sim$ 32 hours (4 workers).
  - Classification takes $\sim$ 12 hours.
  - Plot the figures takes $\sim$ 1 hour.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL-3.0.
- **Archived (provide DOI)**?: https://doi.org/10.5281/zenodo.14811734

4 Artifact Description

4.1 Delivered

https://doi.org/10.5281/zenodo.14811734

4.2 Hardware Dependencies

Any x86-64 machine with at least 64 GB of RAM memory.

4.3 Software Dependencies

Ubuntu, Anaconda.

4.4 Data Sets

- OpenJudge: OpenJudge dataset.
- CC2025: Preprocessed OpenJudge dataset.

5 Artifact Installation

1. Create the directory CC-Artifact.

```
$ mkdir CC-Artifact
```

2. Download and unpack CC2025.tar.xz, Dataset.tar.xz, and Rouxinol.tar.xz from https://doi.org/10.5281/zenodo.14645411, inside CC-Artifact.

```
$ cd CC-Artifact
$ tar xfJ [CC2025 | Dataset | Rouxinol].tar.xz
```

3. Download and install Anaconda.

```
$ ./Anaconda3-2024.10-1-Linux-x86_64.sh -p <CONDA DIRECTORY>
```

4. Create a Rouxinol conda environment.

```
$ cd /<...>/CC-Artifact/Rouxinol
$ conda env create -f rouxinol_<cpu|gpu>.yml
```

5. Activate Rouxinol conda environment.

```
$ conda activate rouxinol
```

6. Install the dependencies

```
$ cd /<...>/CC-Artifact/Rouxinol
$ ./install.sh <CONDA DIRECTORY> <NUMBER OF WORKERS>
```

7. Install Rouxinol.

```
$ cd /<...>/CC-Artifact/Rouxinol
$ pip install .
```

6 Experiment Workflow

You need to set the following variables. ROUXINOL OUTPUT DIRECTORY (ROD) is the directory where the data (LLVM IR, executable, representations, and statistics) will be stored. ROUXINOL DIRECTORY (RD) is Rouxinol's installation directory. CONDA DIRECTORY (CD) is the installation directory of Anaconda. ROUXINOL_APP_DIR (RAD) is the directory that contains the benchmarks; it is an optional variable, and if it is not set, Rouxinol will use the default directory (see Rouxinol/rouxinol/dataset.py) to download and install the benchmarks. Always use the absolute path.

You can use preprocessed data, eliminating the need to generate new data (step 2 can be skipped). To use preprocessed data, ROD should be /<..>/CC-Artifact/CC2025.

1. Activate Rouxinol conda environment.

```
$ conda activate rouxinol
```

2. Generate the data: LLVM IR, executable, representations, and statistics.

```
$ cd /<...>/CC-Artifact/Rouxinol/artifact
$ ./data_generator.sh --out-dir <ROD> --rouxinol <RD> --conda <CD>
  --app-dir <RAD>
```

3. Open games.sh and edit the variables ROD and RD.

```
$ cd /<...>/CC-Artifact/Rouxinol/artifact
$ vim games.sh
```

4. Generate classification results.

```
$ cd /<...>/CC-Artifact/Rouxinol/artifact
$ ./games.sh
```

5. Open plot_figures.sh and edit the variables ROD and RD.

```
$ cd /<...>/CC-Artifact/Rouxinol/artifact
$ vim plot_figures.sh
```

6. Plot the figures.

```
$ cd /<...>/CC-Artifact/Rouxinol/artifact
$ ./plot_figures.sh
```

The folder <ROD>/statistics contains the Figures 4-8.