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

Anderson Faustino da Silva
UEM
Maringá, Brazil
afsilva@uem.br

Jeronimo Castrillon
TU Dresden and SCADS.AI
Dresden, Germany
jeronimo.castrillon@tu-dresden.de

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

## 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.

***CCS Concepts:*** • **Software and its engineering** → **Compilers**.

***Keywords:*** Classification, Valgrind, Binary Diffing

**ACM Reference Format:**
Anderson Faustino da Silva, Jeronimo Castrillon, and Fernando Magno Quintão Pereira. 2025. A Comparative Study on the Accuracy and the Speed of Static and Dynamic Program Classifiers. In *Proceedings of the 34th ACM SIGPLAN International Conference on Compiler Construction (CC '25), March 1–2, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3708493.3712680

## 1 Introduction

Task classification is a well-known computer science problem: given a set of different candidate tasks, and a program that solves one of these tasks, a classifier must determine the correct task that the program implements. This problem is important because it is a core component in malware identification and plagiarism detection [30]. An exact solution to task classification is undecidable, for it amounts to proving equivalence between programs [8, 34]. Thus, prevailing solutions to this challenge are heuristic in nature: a classifier typically uses static program features (sequences of tokens, shape of the abstract syntax tree, histograms of opcodes and such) to categorize programs.

Given the importance of task classification as a central element in the implementation of code diffing systems, many different techniques have been proposed to deal with aspects of this problem. Most of these techniques are purely static: they rely on program features to determine the algorithm that a program implements. These features can be extracted either from the program's source code or from its binary representation. However, the literature also describes task classifiers that use dynamic information; that is, data extracted from the observation of a program's running behavior. For instance, Pewny et al. [31] run the program, and use pairs of inputs and outputs (at the basic block level) to match program points. While accurate, techniques such as Pewny's are considered impractical. Quoting Eschweiler et al [17]: "*While the use of semantic similarity delivers precise results, it is too slow to be applicable to large code bases*". Similarly, Feng et al. [19] would say that: "*The approach by Pewny et al. can take up to one CPU month to prepare and conduct a search in a stock Android image with 1.4 million basic blocks.*"

***Revisiting Dynamic Analyses.*** This paper evaluates the viability of employing dynamic analyses as the foundational technique for developing practical tools to classify programs, utilizing readily available open-source technology. To this end, it compares such analyses with their static counterparts, reporting on the accuracy of both techniques. From this analysis, we report two main results:

1. Dynamic classification can be engineered to be practical: instead of tracking program state (with a reported overhead of thousands of times [15]), it can track only instructions fetched during execution (with an observed overhead of 100x instead of 1000x [15]).
2. A dynamic classifier, even though it has no access to a program source code, can be more accurate than a static classifier, when they use similar representations: the instructions that run vs the instructions that constitute the executable file.

This paper analyzes static and dynamic classifiers that utilize a program's control-flow graph (CFG)—whether in a static form or a dynamic slice from actual execution—to answer the following research question:

> *What is the relative precision and speed of task classifiers that access a static versus a dynamic view of the same program representation?*

While static control-flow graphs can be generated by a variety of tools, constructing dynamic CFGs is considerably more complex. To achieve this, we use CFGGrind [35], a Valgrind plugin. Regarding the embedding technique, various methods exist in the literature for extracting program features from CFGs to enable classification. This paper focuses on opcode histograms; however, in Section 2.2, we demonstrate that histograms are not worse (and often are better) in classification accuracy to previously studied embeddings, such as IR2Vec[36], Inst2Vec[4], and ProGraML [10].

***Summary of Contributions.*** This paper introduces a number of findings, which we enumerate below:

- **Methodology**: the core contribution of this paper is a methodology to apply dynamic analyses to classify programs according to the task they solve. Said methodology requires executing a program, but does not require its source code. Execution is instrumented via CFGGrind [35], a Valgrind [28] plugin.
- **Analyses**: By comparing the accuracy of a purely static and purely dynamic approaches, we observe that the latter outperforms the former marginally in symmetric settings, and largely in asymmetric settings. The asymmetric setup involves an adversary capable of transforming a program before challenging the classifier. Increasing the optimization level used to produce executables enhances the accuracy of dynamic classifiers, although it has only marginal effect on static ones.
- **OpenJudge Dataset**: as a consequence of this study, we have released a public repository with a large dataset of executable programs [1]. Currently, it contains 700 classes of tasks, each with at least 500 programs. This repository is a contribution in itself, for a major part of the authors' time was dedicated to curating the dataset. These programs were extracted from CodeNet [32], filtering out programs that either cause compilation crashes, lack inputs, crash at running time, or do not terminate.

[1]https://github.com/ComputerSystemsLaboratory/Rouxinol

As a result of this work, Section 4 describes Rouxinol, a tool able to classify executable codes by observing traces of instructions. When classifying programming marathon codes compiled with `clang -O3`, given 100 possible candidates, Rouxinol achieves an accuracy of 98%. This accuracy drops to 30% once it is given programs obfuscated with `o-llvm` [24]. Although low, this precision is still more than twice that from a histogram-based classifier that uses static features instead of dynamic observations. Furthermore, we show that the dynamic classifiers either match or outperform tools recently designed to solve code similarity tasks, which require source code, such as IR2Vec or ProGraML. Notice that the dynamic classifier does not need access to a program's source code, nor does it require debugging information in the binary that it analyzes. Although it requires the ability to run programs, it does not observe either the inputs or the outputs of them. Rouxinol and its companion dataset are publicly available under the GPL 3.0 license.

## 2 Static, Dynamic and Hybrid Classifiers

As mentioned in Section 1, the Task Classification problem asks for the task that a program implements, given a set of possible candidates. In this paper, we define "tasks" as problem specifications consisting of pairs of inputs and outputs, following typical online programming judges. Definition 2.1 formalizes this concept.

**Definition 2.1** (Task). A "Task" is a 4-element tuple $\langle D, I, O, C\rangle$, where $D$ is a brief description of the problem that must be solved; $I$ is an ordered list of $n, n > 0$ inputs and $O$ is an ordered list of outputs, such that the output $O[i], 1 \leq i \leq n$ corresponds to the input $I[i]$; and $C$ is a list of constraints that applies to the inputs.

Definition 2.1 is adopted in a large number of previous works that deal with task classification. To illustrate this problem, we reuse Example 2.2, which was taken from the work of Nogueira and Medeiros [29].

**Example 2.2.** (Nogueira and Medeiros [29]) The following task specification was taken from the Code Submission Evaluation System (CSES):

**Description:** Given an array of $n$ integers, find two values at distinct positions, whose sum is $x$.
**Input:** Integer $n$, integer $x$, and array of integers $[a_0, a_1, \ldots, a_{n-1}]$.
**Output:** Integers $0 \leq i < j < n$, such that $a_i + a_j = x$.
**Constraints:** $1 \leq n \leq 2 \times 10^5$; $1 \leq x \leq 2 \times 10^9$; and $1 \leq a_i \leq 2 \times 10^9, 0 \leq i < n$.

## 2.1 Binary Program Diffing

Task classification is a form of *Program Diffing*, a very broad computer science challenge whose goal is to determine how similar two programs are. Establishing the equivalence of two programs presents an undecidable challenge, even within the constraints of basic formal language classes [23]. Consequently, the majority of algorithms designed to assess program similarities rely on heuristics. The evolution of these algorithms has a rich history, with early discussions centering around similarity at the source code level. Notably, Hunt et al.[22] suggested, as early as 1977, that their sequence alignment algorithm could be applied to ascertain program similarity.

In this paper, we use task classification techniques to solve a variation of code diffing called *Binary Diffing*: thus, the input of our task classifiers are programs written in binary format. Binary diffing began gaining recognition in the nineties. The algorithms proposed during this period were often based on dynamic programming techniques, such as sequence alignment [3, 9], or on hash functions to match code blocks [38]. However, more recent approaches have drifted towards stochastic techniques, typically relying on machine-learning models to indicate a probability that two programs implement the same task. In this regard, the number of different techniques to solve binary diffing is vast. As a testimony to this variety, the "*Awesome Binary Similarity*" website [25] listed 213 publications related to the topic in November of 2024. Nevertheless, all these stochastic techniques follow a very specific pattern, which Figure 1 describes.

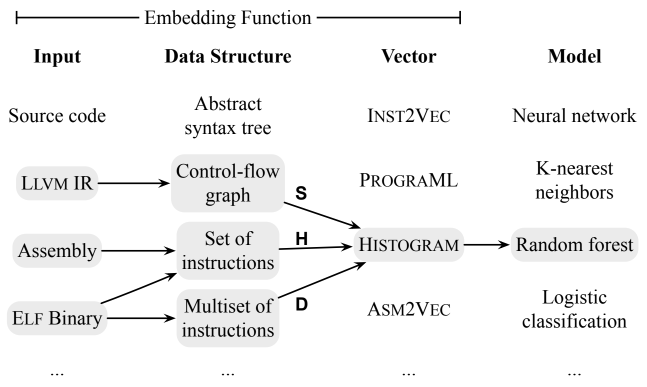

**Figure 1.** Overview of different approaches to solve algorithm classification. The techniques studied in this paper appear in gray boxes.

## 2.2 Static/Hybrid/Dynamic Classifiers

Most of the over 200 implementations of code classifiers listed by Song Liu [25] implement *embedding functions*: functions that map programs onto a vector space. These vectors are then compared—e.g., via their Euclidean Distance—to determine if they represent similar programs. One of the most common types of vectors are *histograms*. As an example, at least three algorithm classifiers [12, 13, 20] released in 2023 compare histogram of instructions extracted from the LLVM intermediate representation of programs. Because this setup is so common in the recent code-diffing literature, we use it as a baseline in this paper. Henceforth, we call such approaches *Purely Static Techniques*, because obtaining histograms of instructions does not require running the program. Figure 1 indicate purely static strategies with an **S**.

In this paper, our focus is on *dynamic classifiers*, which are granted the ability to observe program execution. Such classifiers prove beneficial in various scenarios, such as identifying plagiarism in online grading systems or detecting binary programs that improperly utilize third-party code. In this regard, we further define two types of classifiers: *Hybrid* and *Purely Dynamic*, which we define as follows:

**Definition 2.3** (Hybrid/Dynamic Classifiers)**.** The execution of a program, for a given input, consists of a *trace of instructions*. This trace is a sequence of pairs $S = ((a_n, o_n))$, where each element is formed by an instruction address ($a$) and an opcode ($o$). A hybrid classifier observes this trace as a *set*; that is, without duplicate elements. It then builds a histogram for the opcodes, with the value for opcode $o'$ defined as $h_{\text{hybrid}}(o') = |\{(a, o) \in S; o = o'\}|$.

A dynamic classifier, in turn, counts every appearance of an opcode in the trace and computes a value for an opcode $o'$ in the histogram as $h_{\text{dyn}}(o') = \sum_{(a,o)\in S, o=o'} 1$.

**Example 2.4.** Figure 2 shows histograms extracted from different program representations. The LLVM intermediate representation of programs, illustrated in Figure 2 (b), yield the histograms seen in Figure 2 (e) that power the fully static classifiers of Definition 2.3. Fully dynamic classifiers, as seen in Figure 2 (d), obtain their information from traces of fetched instructions. The trace in Figure 2 (d) was produced by the command line `./a.out a a`, where `a.out` is the executable compiled from Figure 2 (a). A histogram derived from such a trace—seen in Figure 2 (g)—is called *fully dynamic*. Finally, hybrid classifiers can observe the portion of a program that is covered by the flow of execution. Figure 2 (c) shows the instructions covered by the execution of `./a.out a a`, and Figure 2 (f) shows the ensuring histogram.

Embedding functions used by hybrid and dynamic classifiers depend on the program's code and inputs. As seen in Example 2.4, a dynamic classifier counts how many times each instruction address has been visited by the program counter. Thus, instructions within loops might contribute more to the histogram than instructions outside loops. A hybrid classifier tracks which instructions where *covered* by the execution flow, without counting how often each instruction was fetched. Example 2.5 clarifies this difference.

**(a)**
```
int main(int a, char** v) {
    int f = 1;
    while (a > 1) {
        f *= a;
        --a;
    }
    return f;
}
```

**(b)**
```
define @main(%a, %v) {
bb:
  br label %bb5
bb5:
  %i2 = phi [%a:%bb], [%i13:%bb8]
  %i4 = phi [1:%bb], [%i11:%bb8]
  %i7 = icmp sgt %i2, 1
  br i1 %i7, %bb8, %bb14
bb8:
  %i11 = mul %i4, %i2
  %i13 = add %i2, -1
  br %bb5
bb14:
  ret %i4
}
```

**(c)**
```
0000000000401110 <main>:
401110: mov $0x1,%eax
401115: cmp $0x2,%edi
401118: jl 40112d <main+0x1d>
40111a: mov $0x1,%eax
40111f: nop
401120: imul %edi,%eax
401123: lea -0x1(%rdi),%ecx
401126: cmp $0x2,%edi
401129: mov %ecx,%edi
40112b: jg 401120 <main+0x10>
40112d: retq
40112e: xchg %ax,%ax
```

**(d)**
```
401110: mov $0x1,%eax
401115: cmp $0x2,%edi
401118: jl 40112d <main+0x1d>
40111a: mov $0x1,%eax
40111f: nop
401120: imul %edi,%eax
401123: lea -0x1(%rdi),%ecx
401126: cmp $0x2,%edi
401129: mov %ecx,%edi
40112b: jg 401120 <main+0x10>
401120: imul %edi,%eax
401123: lea -0x1(%rdi),%ecx
401126: cmp $0x2,%edi
401129: mov %ecx,%edi
40112b: jg 401120 <main+0x10>
40112d: retq
40112e: xchg %ax,%ax
```

**(e)** Static Static Histogram

| br | phi | icmp | mul | add | ret |
|---|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 1 | 1 |

**(f)** Hybrid Histogram: `a.out a a`

| mov | cmp | jl | nop | imul | lea | jg | ret | xchg |
|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

**(g)** Fully Dynamic Histogram: `a.out a a`

| mov | cmp | jl | nop | imul | lea | jg | ret | xchg |
|---|---|---|---|---|---|---|---|---|
| 4 | 3 | 1 | 1 | 2 | 2 | 2 | 1 | 1 |

**(h)** Hybrid Histogram: `a.out`

| mov | cmp | jl | nop | imul | lea | jg | ret | xchg |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

**(i)** Fully Dynamic Histogram: `a.out`

| mov | cmp | jl | nop | imul | lea | jg | ret | xchg |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

**Figure 2.** (a) Program written in C. (b) LLVM intermediate representation. (c) x86 Assembly. (d) Trace of instructions fetched during execution. (e) Histogram of LLVM instructions (static). (f,h) Histogram of set of visited instructions (hybrid). (g,i) Histogram of instruction executions (dynamic).

**Example 2.5.** Figures 2 (f) and (h) show two hybrid histograms. They differ because they were produced by different inputs. The former was generated by the command line `./a.out a a`; the latter, by `./a.out` without any argument. If the target program is invoked without arguments, then the instructions between addresses `40111a` and `40112b` are not visited by the execution flow. Thus, none of these instructions will contribute to the histogram. Fully dynamic histograms also vary, depending on program inputs, as Figures 2 (g) and (i) demonstrate.

Hybrid and dynamic histograms can be reconstructed by observing program executions, and there are several publicly available tools that facilitate such observations, including PIN [26], DynamoRIO [7], or Valgrind [28]. In this paper, we employ CFGGrind [35], a Valgrind plugin that reconstructs control-flow graphs (CFGs) of executable programs. Only instructions covered by the execution flow contribute to these CFGs. The opcodes in the CFGs provide us with the hybrid histograms. The graphs produced by CFGGrind include execution counts on the edges, which we use to reconstruct the dynamic histograms.

## 3 Classification Methodology

This paper has two goals. First, to show how the extra information accessible to a dynamic algorithm classifier improves its accuracy when compared to a fully static technique. Second, to show that collecting this information can be done within reasonable time. To this end, this section introduces the methodology that we have employed to perform task classification. This methodology uses a dataset of executable programs discussed in Section 3.1 as inputs to the different *comparison games* explained in Section 3.2.

As explained in Section 2, this paper recognizes three different ways to obtain the histogram of instructions of a program: static, hybrid and dynamic. Figure 3 shows how we obtain each of these kinds of histograms. Notice that three of the histograms in Figure 3 count x86 instructions, whereas one of them counts LLVM instructions. The x86 format is the most common source of information among the more than 200 works listed by [25]. In this paper, we also analyze the LLVM because it has been used in recent works on task classification [6, 13, 20].

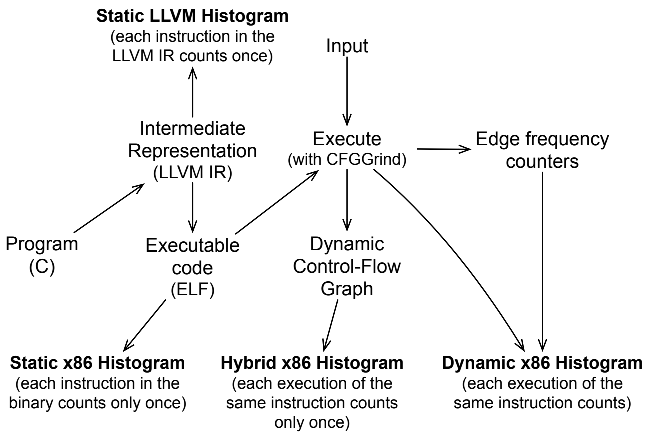
Static LLVM Histogram
(each instruction in the LLVM IR counts once)
Input
Intermediate Representation (LLVM IR)
Execute (with CFGGrind)
Edge frequency counters
Program (C)
Executable code (ELF)
Dynamic Control-Flow Graph
Static x86 Histogram
(each instruction in the binary counts only once)
Hybrid x86 Histogram
(each execution of the same instruction counts only once)
Dynamic x86 Histogram
(each execution of the same instruction counts)

**Figure 3.** The process of generating different histograms of opcodes. The accuracy of classifiers based on these different histograms shall be compared in Figure 5 (Page 7).

### 3.1 Dataset

In this paper, we evaluate different techniques to solve algorithm classification using C programs taken from the OpenJudge repository [32]. Our experiments require executable programs; however, programs in OpenJudge are not distributed in a way that facilitates execution: the repository contains code that does not compile with the compilers we used; some programs lack inputs; others run with errors. Thus, we have filtered programs that satisfies the following requirements:

1. Each program can be compiled with `clang` 10.0.0 and transformed with `o-llvm` [24].
2. Each program runs without errors (exit code `0`) with **all** the available inputs on an Intel i7.

Out of this subset of OpenJudge, we built a collection of 700 classes of programs, each class containing at least 500 programs. All programs of a class solve the same instance of a programming marathon problem. From this notion, we can define the Algorithm Classification problem as follows:

**Definition 3.1** (Alg. Classif.). Let $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ be a collection of sets of known programs, such that any $p \in F_i$ solves the same task $i$. Given a set of known program classes $\mathcal{F}$ and a candidate program $p$, algorithm classification consists in determining the task $j, F_j \in \mathcal{F}$ that $p$ implements.

Each class of problem in the OpenJudge suite comes with exactly one input and its corresponding expected output. We ensure that all programs in the same class produce equal outputs for that input. However, these programs are not guaranteed to be equivalent. Thus, it is possible that they might differ for other inputs not present in our problem description, either producing different outputs, crashing or looping indefinitely. Therefore, in the experiments reported in Section 4, we use one input per benchmark to build that benchmark's histogram. Notice that this methodology is sound, as the same class of problems is either used as part of a training set, or as part of a test set, but never in both.

### 3.2 Adversarial Games

This paper compares algorithm classifiers using the *Game Framework* proposed by Damásio et al [13]. A game pitches a *classifier* against an *evader* on an algorithm classification problem, following a three-stage protocol, defined as follows:

**Definition 3.2** (Classification Game). A game is formed by a classifier $C$, an evader $E$, a collection of program classes $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ plus their inputs $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$, where each $I_i$ is a collection of inputs[2] for programs in $F_i$. A game happens in three phases.

- **Training:** The classifier $C$ observes the set of program classes $\mathcal{F}$ and their inputs $\mathcal{I}$ to build a classification model trained with data from $\mathcal{F}$ and $\mathcal{I}$.
- **Challenge:** The evader $E$ draws a new program $p$ from the same distribution used to generate the training set for the classifier. The classifier has not seen $p$ in the training phase. $E$ is allowed to apply onto $p$ a semantics-preserving transformation $f_e$; thus, obtaining an equivalent program $p' = f_e(p)$.
- **Response:** The classifier receives $p'$, and might further modify it using a semantics-preserving transformation $f_c$; thus, obtaining a program $p'' = f_c(p')$. $C$ might observe the execution of $p''$ on $I_j, 1 \leq j \leq n$. It must guess the class $F_i, 1 \leq i \leq n$, to which $p''$ belongs.

Depending on which transformation functions ($f_e$ and $f_c$) are available for evaders and classifiers, games can either be *symmetric* or *asymmetric*. Symmetric games give classifiers and evaders the same function; asymmetric games give them different transformations:

- **Game-0** is a symmetric game where $f_c = f_e =$ identity
- **Game-1** is an asymmetric game where $f_c =$ identity and $f_e \neq$ identity.
- **Game-2** is a symmetric game where $f_c = f_e \neq$ identity.

Early discussions of algorithm classification would evaluate instances of **Game0** [1, 4, 27]. Research that focuses on building evaders tend to adopt methodologies centered around **Game1**. Such is the case of the work of Ren et al. [33], which created the well-known BinTuner code modifier. In the case of BinTuner, $f_e$ is a sequence of code optimizations. The literature usually does not focus on **Game2** on its own, as it is considered a direct variation of **Game0**. However, Damásio et al. report that some obfuscation techniques might reduce the precision of an algorithm classifier. Results in Section 4 corroborate this observation when $f_e = f_c =$ `o-llvm`, where `o-llvm` is Junod et al.'s LLVM-based obfuscator [24]. In this regard, the experimental methodology used in this paper adopts different code transformation techniques to evaluate instances of **Game1** and **Game2**:

- `fla`: Control-Flow Flattening, which replaces the CFG of the program with a switch within a loop. Each case of the switch is a basic block in the original program.
- `bcf`: Bogus Control Flow, which inserts new execution paths into a program's control-flow graph. These new paths are unreachable, for they are guarded by conditions that are always false. These conditions are engineered in a way to remain in place in spite of compiler optimizations.
- `sub`: Instruction Substitution, which replaces sequences of instructions with code of similar semantics.

***Classification Models.*** Although the four different histograms in Figure 3 come from different program representations, they can be used in tandem with the same classification model adjusted for different vector lengths. In other words, the different histograms are all $n \times 1$ vectors, where $n$ is the number of opcodes in the given representation: 66 for LLVM and more than 1,000 for x86 (however, in our experiments, only a few of these opcodes are observed: approximately 300). In this paper, we perform classification via random forests.

## 4 Evaluation

Our evaluation breaks down the main research question introduced in Section 1 into:

**RQ1:** What is the accuracy of the classifiers evaluated in this paper compared to either trivial classifiers or to state-of-the-art techniques discussed in the literature?

**RQ2:** What is the relative accuracy of purely static, hybrid and purely dynamic task classifiers?

[2]Definition 3.2 assumes that data inputs exist for each task; however, it does not assume that reference outputs are available, nor that inputs are distinct.

**RQ3:** How does the optimization level used in compilation influence the accuracy of the different classifiers?
**RQ4:** How does code obfuscation influence the classifiers, when the classifier is aware and unaware of the obfuscation technique used?
**RQ5:** How much time is necessary to carry out classification using the techniques evaluated in this paper?

We evaluate **RQ1** to provide perspective on the results that this paper reports: by comparing the different classifiers from Section 2 with well-known baselines, we show that the experimental setup discussed in this paper is competitive with state-of-the-art algorithm classifiers.

***Experimental Setup.*** Results discussed in this paper were produced on an AMD Ryzen Threadripper 3960X 24-Core Processor 2.2 GHz, with 64 GB of RAM, running Linux Ubuntu v20.04. Every code in the dataset of 500 × 100 programs was compiled with clang v10.0. The dynamic and hybrid classifiers use CFGGrind, commit `d3fcdd3`, which runs on Valgrind v3.22.0. In every game, the classifier was trained with 80% of the dataset and tested with the remaining 20%. Obfuscation techniques come from `o-llvm` [24] (commit `8bd80ea`, LLVM 10.x). The dynamic and the hybrid histograms of a benchmark are constructed after observing one execution of that benchmark, i.e., with one input. Classification via the embedding functions discussed in Section 2 use a standard implementation of random forest taken from `SciKit-Learn`. This model is also used in Section 4.1, when evaluating IR2Vec—in our setting, we have obtained better results with the random forest than using IR2Vec's original neural network [36]. We evaluate ProGraML using Brauckmann et al. [5] implementation of a graph neural network available in CompPy-Learn.

## 4.1 RQ1: Comparison with Previous Work

The classifiers assessed in Sections 4.3 ßnd 4.4 have not undergone testing in prior studies. Consequently, to offer the reader comparison points, this section contrasts these classifiers with two sets of baselines: trivial and state-of-the-art. The latter group features the following previous work:

- **ProGraML:** a graph-based embedding function that maps elements of the program's control-flow graph to a 32 × 64-bit vector [10].
- **IR2Vec:** a flow-sensitive embedding that maps program instructions to a 300 × 64-bit vector [36].

Additionally, we consider three trivial classifiers, based on the size of histograms of opcodes. By evaluating trivial classifiers we demonstrate that the effectiveness of the embedding functions discussed in Section 2 does not come from counting the instructions that make up programs. Thus, each trivial classifier is extracted from one of the embeddings seen in Section 2: The S-Trivial classifier uses the LLVM Static representation; the H-Trivial classifier uses the CFGGrind Hybrid representation; and the D-Trivial classifier uses the CFGGrind Dynamic representation. We call the *size* of a histogram the sum of all the opcodes that it contains. As an example, the size of the histogram in Figure 2 (e) is 9, and the size of the histogram in Figure 2 (f) is 12. Each trivial classifier averages the sizes of histograms per class of problem. When given an unknown problem, that classifier matches the size of its histogram with the class of closest average.

***Discussion.*** Figure 4 compares the different embedding functions with respect to Game-0. For each embedding, this experiment uses either programs compiled without optimizations (clang -O0) or programs compiled with the maximum optimization level of clang: -O3. This section uses a reduced number of classes of algorithmic problems: 32 classes with 500 samples each. Notice that in the next sections we use 100 classes with 500 samples each. The reduction in the number of classes was necessary because IR2Vec is very memory-intensive: we could not set it up in our environment using more problems, or with more samples per problem.

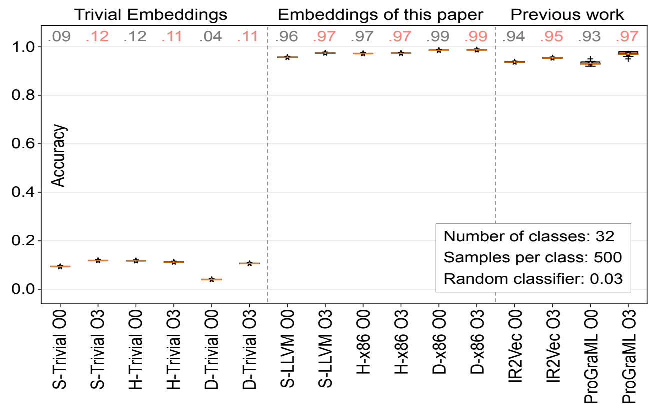

**Figure 4.** A comparison of different embedding functions on Game-0.

Figure 4 makes it clear that using the trivial classifiers is only marginally better than matching programs randomly. Accuracy ranges from 4 to 12%, while the expected accuracy of a random classifier is 3% (1/32). This low performance is in contrast with the accuracy of the other classifiers, which is always above 90%. Thus, the classifiers evaluated in Sections 4.3 and 4.4 are not simply counting the number of opcodes observed during the execution of programs—they are exploring relations between these opcodes. Another notable finding from Figure 4 is that the embeddings assessed in this paper demonstrate comparable performance to established program representations commonly employed for binary diffing. Specifically, the embedding functions from Section 2 either surpass or achieve similar results to IR2Vec and ProGraML. This performance suggests that the observations to be discussed in the next sections come from a robust experimental setup, aligning with current state-of-the-art approaches for addressing code similarity challenges.

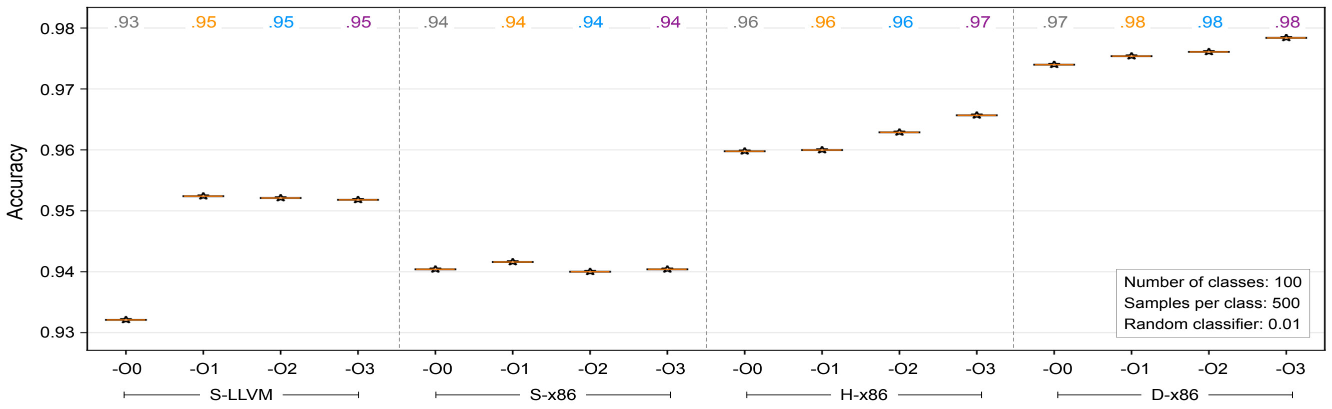

**Figure 5.** The impact of code optimizations on the accuracy of classifiers.

## 4.2 RQ2: Static, Hybrid and Dynamic Classifiers

Most previous work on task classification are purely static: they do not run the target programs. Static information can be extracted at the source or at the binary level. Source-code embeddings can be extracted from the sequence of tokens [6], the abstract syntax tree [2] or even the compiler's intermediate representation of a program [12]. The literature on binary diffing focus, evidently, on binary level data. In this case, the information necessary to classify programs is extracted from the object representation of those programs. In this section, we compare the relative accuracy of these pure static techniques with the dynamic approaches that we advocate in this paper. To this end, we consider two static classifiers, which have been recently discussed in previous work. One of them, evaluated in the work of da Silva et al. [12] compares histograms extracted from the LLVM representation of programs. The second classifier, recently presented by VenkataKeerthy et al. [37], extract histograms from the x86 binary representation of programs, produced via the `Capstone` Disassembler v5.0 [18].

***Discussion.*** Figure 5 compares the accuracy of the different classifiers. The figure lets us draw a few conclusions:

1. The dynamic classifiers are strictly better than the static ones. This result is statistically significant: a t-test performed on a population drawn from a static technique and another drawn from a hybrid/dynamic technique always yields a p-value close to zero.
2. At higher optimization levels, the LLVM IR yields more accurate data than the x86 binary. Without optimizations, the LLVM variables are mapped onto memory, and the excess of loads and stores pollutes the histograms. At `-O1`, the `mem2reg` pass maps variables to virtual registers, eliminating most of the memory-access operations.
3. Optimizations have a positive effect on hybrid and dynamic classifiers, but not on the static approaches. Section 4.3 further discusses this observation.

All the classifiers analyzed in Figure 5 produce non-trivial results. The expected accuracy of a random classifier is 1%. In contrast, any of the histogram-based approaches yields accuracies superior to 90%. However, there is a clear advantage on hybrid/dynamic classification over static techniques. The Valgrind-based approaches capture the instructions that are actually executed during a run of the program. Since they focus on runtime behavior, they can filter out dead code. This results in higher accuracy because only instructions that truly run are actually compared—something that correlates well with how similar programs behave during execution.

We observe a slight advantage of the dynamic classifier over the hybrid technique. Counting the number of times each opcode was visited allows the dynamic classifier to capture dynamic behavior with even more granularity, particularly in terms of loop structures and frequently executed paths. This methodology is more effective for detecting similarity because it reflects not just the static instructions, but also the actual workload and runtime behavior of the program. Therefore, programs with similar structures but different runtime behaviors (e.g., different loop bounds) might stand out more clearly in this setup.

## 4.3 RQ3: On the Impact of Code Optimizations

Previous work has observed that optimizations improve the accuracy of code classifiers [13, 14]. As an example, Damasio et al. [13] have shown that clang -O1 reverses almost every effect of the obfuscation techniques proposed by Zhang et al. [39]. Similarly, VenkataKeerthy et al. [37] have recently shown that optimizations can be employed as a normalization technique to simplify tasks of code classification. This section analyzes if such results remain true for classifiers that use dynamic execution traces.

***Discussion.*** Figure 5 analyzes the accuracy of different code classifiers once they are trained and tested with programs compiled with different levels of optimization. The data in Figure 5 refers to *symmetric games*: the classifier is trained using programs compiled with the same flags that the evader uses. Regardless of the program representation used, or of the optimization level adopted, all the classifiers of Figure 5 have accuracy above 90%—well above the expected accuracy of a random classifier, which would be 1%. Thus, they are able to learn code properties.

The only setting where optimizations have improved algorithmic classification concerns the histogram of LLVM opcodes. This result is on par with those reported by Damasio et al. [13], or by Gorchako et al. [20]. We notice that in the LLVM setting, the simple fact of keeping program variables in virtual registers (instead of mapping them to memory) already reduces the size of programs by almost half. We speculate that the part of the program that remains—logical, arithmetic and control-flow instructions, for instance—contains more useful semantic information. This behavior does not occur in the x86 setting: since there are few general purpose registers, most of the program data is accessed via `mov` instructions, regardless of the optimization level. Nevertheless, Figure 5 shows a consistent accuracy improvement as we move from the LLVM purely static representation to x86 hybrid histograms, and then to x86 dynamic histograms.

## 4.4 RQ4: On the Impact of Code Obfuscation

It is widely acknowledged that code obfuscation techniques can effectively evade algorithmic classification, as demonstrated by various studies [13, 21, 33]. Numerous obfuscation methods exist, with some involving the insertion of dead code into programs. These seemingly innocuous instructions are crafted to withstand traditional compiler optimizations. Yet, dead-code does not run; consequently, dynamic program representations like H-x86 and D-x86 are expected, in principle, to resist this type of obfuscation. This section explores whether this expectation holds true.

***Discussion.*** Figure 6 compares the classifiers of Section 2 in four different settings. The three first settings use the obfuscation techniques mentioned in Section 3.2. The fourth setting, `all`, combines the three obfuscation techniques. In almost every scenario, transitioning from LLVM histograms to hybrid x86 histograms and then to dynamic x86 histograms enhances classification accuracy. The only exception to this trend in Figure 6 is observed when transitioning from `S-LLVM fla` to `H-x86 fla`, where accuracy drops from 77 to 60%. We speculate that this decrease in accuracy happens because LLVM still preserves explicit control flow: a switch is represented as an actual LLVM instruction, whereas in x86 the switch becomes an indirect jump whose address is loaded from an array of addresses. Thus, out of all the different obfuscation techniques, control-flow flattening causes the largest difference between the LLVM and the x86 representations. Sometimes the improvement in accuracy, when moving from a purely static to a purely hybrid approach, is substantial. For example, considering **bcf**, the accuracy of a purely dynamic classifier (D-x86 bcf) is more than twice as high as the accuracy of a purely static one (S-LLVM bcf).

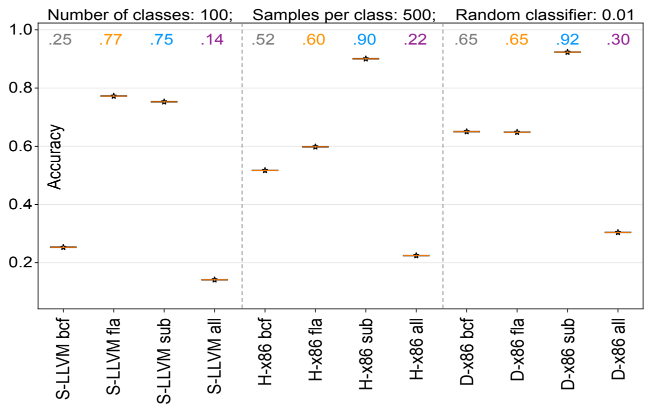

**Figure 6.** The impact of code obfuscations on the accuracy of the different classifiers on **Game-1**: an asymmetric game, where the classifier is not aware of the obfuscator used by the evader.

These findings supports the intuition that dynamic analyses are expected to be more resilient to obfuscation techniques that insert dead code in programs such as bogus control flow. Notice that these techniques still bring some effectiveness, for they create conditionals that are executed. Furthermore, histogram-based classifiers, as already previously observed [13], tend to resist transformations such as flattening, which change control flow, but preserve the mix of instructions. However, code obfuscation remains effective in concealing the purpose of programs. In the asymmetric game where the evader uses all the available obfuscations, the accuracy of our best classifier—the purely dynamic approach—is only 30%. While this accuracy is modest, it surpasses the accuracy of a purely static approach, which stands at 14%.

Figure 6 shows results for an asymmetric game: the classifier is trained with unmodified programs, and tested with obfuscated codes. If the classifier is trained with obfuscated programs, its accuracy improves substantially. Figure 7 supports this statement with data. In this scenario, the same obfuscation technique is applied to both the training and testing sets. Consistent with the trends observed in Figures 5 and 6, accuracy improves across the spectrum from the purely static to the purely dynamic approach. While obfuscation continues to impact the accuracy of all classifiers, the most severe degradation occurs in the purely static scenario. The accuracy of the LLVM-based classifier drops from 94% to 69% when applied to the fully obfuscated dataset.

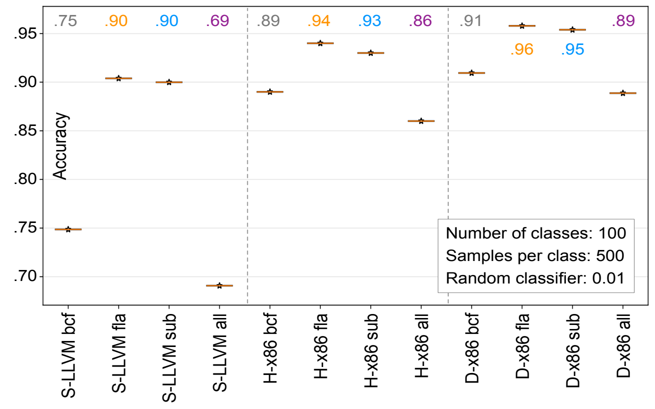

**Figure 7.** The impact of code obfuscations on the accuracy of the different classifiers on **Game-2**: a symmetric game, where the classifier is trained with the obfuscation techniques used by the evader.

## 4.5 RQ5: Running Time

Since the four techniques analyzed in Figure 6 utilize opcode histograms, the computational costs of training these models and classifying new histograms are very similar. However, the overhead associated with extracting these histograms varies significantly. Producing histograms of LLVM instructions (`S-LLVM`) requires converting C programs into LLVM IR. Similarly, obtaining histograms of x86 static instructions (`S-x86`) involves the costs of assembling and linking object files into executables. Finally, both CFG-grind-based approaches (`H-x86` and `D-x86`) entail the overhead of executing an instrumented version of each program using Valgrind. This section compares these different overheads.

***Discussion.*** Figure 8 illustrates the overhead of various approaches to obtain opcode histograms. All four approaches incur the same cost to produce the LLVM intermediate representation of programs (dark-gray bars at the bottom). This cost naturally varies with the optimization level: using `-O0` as a baseline, we observe virtually no time difference with `-O1`, but a slowdown of 1.4x at `-O2` and 2.1x at `-O3`. The extraction of x86 histograms (middle-gray bars) requires generating an executable[3], and this process incurs similar costs across `-O1`, `-O2`, and `-O3`: approximately 1.3x lower than at `-O1`, due to the assembler handling smaller code sizes. Notice that `S-LLVM` does not pay this cost. Finally, the cost of histogram extraction (light-gray bars on top) increases substantially when program execution is required: in our setup, executing the programs with CFGGrind is about 70x slower than using Capstone to parse the program’s assembly and nearly 140x slower than using LLVM’s `opt` to process the LLVM IR. This cost depends directly on the execution time of each program, and hence varies with both the program itself and its inputs. However, once we consider the entire process—from the construction of the program representation to the extraction of histograms—if `H/D-LLVM` serves as the baseline, then `S-x86` is 8.8x faster, while `S-x86` is 4.0x faster.

[3]In a realistic use-case scenario, the tester is likely to be given the executable; hence, in practice, the two lower bars often will not be perceived as costs.

## 5 Related Work

The literature on binary diffing recognizes two distinct approaches to compare programs: dynamic and static. The former relies on data observed during program execution, while the latter utilizes the program’s syntax. In recent years, the static approach has gained more popularity. To support this claim, we refer the reader to the repository curated by Song Liu [25], which catalogs over 220 publications on binary similarity. All ten publications from 2023 in the repository utilize program instructions (syntax) rather than program state (semantics) as the basis for solving binary similarity. Haq and Caballero [21] suggest that while research on static diffing techniques emphasizes accuracy, research on dynamic approaches prioritizes coverage and practicality.

***Dynamic Analyses for Binary Diffing.*** As mentioned in Section 1, some of the initial attempts to address the binary similarity problem, such as Pewny et al.’s [31], were dynamic in nature. This category of work deals with *Behavioral Equivalence*. Essentially, the *program state* provides the data for comparing binaries. As a simplification, we refer to the program state as the load of the *Data Cache*; encompassing the contents of the heap, stack, globals, and string literals. This characteristic of prior solutions to binary diffing contrasts with the approach taken in this paper, where we exclusively use the contents of the *Instruction Cache* to compare programs.

Examining the program state is computationally expensive. For instance, a recent study by Wesley et al. [16] reveals that the time to dump the contents of the program heap, exclusively at the end of execution, increases running time by two orders of magnitude for the MiBench benchmark suite. Notice that this result regards one dump. In de Souza Magalhães et al.’s word, “*With only 267 LLVM instructions,* `dijkstra` *is the second smallest benchmark in the MiBench suite. However, it stores a large quantity of data in the heap: even printing this data at every static inspection point is not practical.*” de Souza Magalhães et al. additionally demonstrate that tracking program state is typically imprecise, whether due to the challenging nature of languages like C or C++, or the difficulty in ensuring comprehensive coverage.

Hence, it is not surprising that recent research on behavioral equivalence of programs mainly focuses on either reducing the overhead of reading program state or increasing code coverage. For a comprehensive overview of the current state-of-the-art techniques in this field, we recommend the recent work of Zhou et al. [40]. The design of Arcturus, the

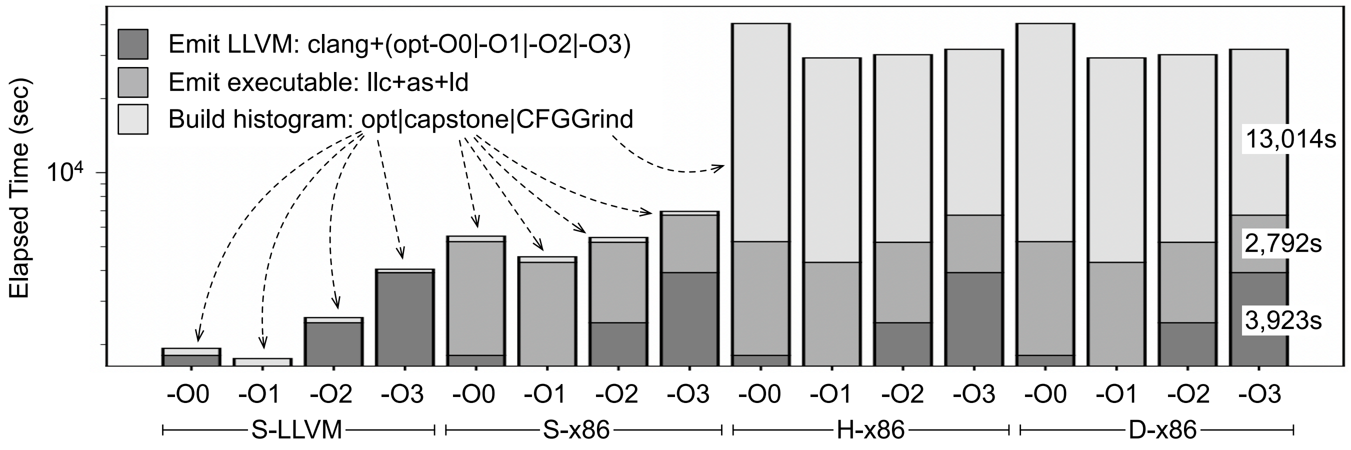

**Figure 8.** Running time of different task classification approaches. We leave numbers in the last bar to give the reader a notion of absolute scale, as the figure uses logarithmic scale.

tool presented in Zhou et al.'s work, adeptly incorporates key techniques for managing cost and coverage: to reduce cost, Arcturus concentrates on specific program paths within a program; to increase coverage, Arcturus can enforce the execution of specific program points. Nevertheless, the reported running times by Zhou et al. indicate that using a tool like Arcturus to analyze the thousands of programs evaluated in Section 4 would not be feasible.

***Static Analyses for Binary Diffing.*** The literature on binary diffing predominantly concentrates on static analyses. The burgeoning popularity of machine-learning techniques has sparked new interest in this field, leading to the release of novel program embeddings every year. For an overview of popular embeddings, we recommend Section 2 of da Silva et al. [12]'s survey. This work evaluates embeddings like histograms of opcodes, which have been used in recent work [13, 20]. However, our approach involves extracting these embeddings from the execution of programs rather than from their static assembly representation. This perspective distinguishes this paper from the literature on static binary diffing. Furthermore, since histograms of opcodes have not been explored in dynamic binary diffing analyses, this work stands out as an original contribution.

## 6 Conclusion

This paper has compared static and dynamic classifiers in terms of precision and speed. Unlike previous dynamic classifiers that relied on observing the state of programs, specifically the values stored in the data cache, this work demonstrates that precise classification can be achieved solely by observing the stream of instructions fetched during program execution. This result is significant because monitoring the instruction stream is a more straightforward task compared to tracking the program state, which often involves traversing the graph of reachable program data. Furthermore, the dynamic classifiers discussed in this paper require neither the availability of source code, nor the presence of debugging information in the binary programs, nor the ability to check the program's output. Beyond its precision, this approach also exhibits increased resilience, albeit not complete immunity, to code obfuscation. Notably, classification based on histograms of fetched instructions tends to better withstand obfuscation based on dead-code insertion. Highlighting a key result, we observe that the histogram-based dynamic classifier is twice as accurate as the static classifier (Sec. 4.4), while being 8.8x slower (Sec. 4.5). Yet, obfuscation, in general, is still a challenge for binary diffing, as the experiments in this paper demonstrate. The craft of algorithms able to identify heavily obfuscated codes is an open question, onto which we shall be working. Nevertheless, we expect that the findings already reported in this paper prove valuable in activities such as copyright auditing, malware identification, plagiarism detection, and any other scenario where accurate identification of program similarities is essential.

## Data Availability Statement

This paper's reproducible artifact is available either at https://github.com/ComputerSystemsLaboratory/Rouxinol or via Zenodo [11].

## Acknowledgment

This project has been sponsored by the following research agencies: CNPq (304441/2021-0 & 444127/2024-0), FAPEMIG (APQ-00440-23), CAPES (PrInt), the Center for Advancing Electronics Dresden (cfaed), and the AI competence center ScaDS.AI Dresden/Leipzig in Germany (01IS18026A-D). We thank Simon Meusel for revising an earlier version of this manuscript.

## References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (jul 2018), 37 pages. https://doi.org/10.1145/3212695

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353

[3] Brenda S. Baker and Udi Manber. 1998. Deducing Similarities in Java Sources from Bytecodes. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*. USENIX Association, New Orleans, LA. https://www.usenix.org/conference/1998-usenix-annual-technical-conference/deducing-similarities-java-sources-bytecodes

[4] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. In *NIPS* (Montréal, Canada). Curran Associates Inc., Red Hook, USA, 3589–3601. https://doi.org/10.5555/3327144.3327276

[5] Alexander Brauckmann, Andrés Goens, and Jeronimo Castrillon. 2020. ComPy-Learn: A Toolbox for Exploring Machine Learning Representations for Compilers. In *FD)* (Kiel, Germany). 1–4. https://doi.org/10.1109/FDL50818.2020.9232946

[6] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *CC* (San Diego, USA). ACM, New York, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *CGO* (San Francisco, California, USA). IEEE Computer Society, USA, 265–275.

[8] B Jack Copeland. 1997. The church-turing thesis. Available at https://plato.stanford.edu/ENTRIES/church-turing/.

[9] Krish Coppieters. [n. d.]. A Cross-Platform Binary Diff. https://www.drdobbs.com/embedded-systems/a-cross-platform-binary-diff/184409550. [Online; accessed 12-Nov-2023].

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. 2021. ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations. In *ICML*, Vol. 139. PMLR, Baltimore, Maryland, USA, 2244–2253.

[11] Anderson Faustino da Silva. 2025. Rouxinol. https://doi.org/10.5281/zenodo.14645411 Accessed: 2025-01-17.

[12] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintão Pereira, Nilton Luiz Queiroz Junior, and Otávio Oliveira Napoli. 2022. Program representations for predictive compilation: State of affairs in the early 20's. *J. Comput. Lang.* 73 (2022), 101171. https://doi.org/10.1016/j.cola.2022.101171

[13] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. 2023. A Game-Based Framework to Compare Program Classifiers and Evaders. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3579990.3580012

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3062341.3062387

[15] José Wesley de Souza Magalhães, Chunhua Liao, and Fernando Magno Quintão Pereira. 2022. Automatic inspection of program state in an uncooperative environment. *Softw. Pract. Exp.* 52, 12 (2022), 2727–2758. https://doi.org/10.1002/SPE.3146

[16] José Wesley de Souza Magalhães, Chunhua Liao, and Fernando Magno Quintão Pereira. 2022. Automatic inspection of program state in an uncooperative environment. *Softw. Pract. Exp.* 52, 12 (2022), 2727–2758. https://doi.org/10.1002/SPE.3146

[17] Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.. In *Ndss*, Vol. 52. 58–79.

[18] Nguyen Anh Quynh et al. 2024. Capstone Engine. https://www.capstone-engine.org/.

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-Based Bug Search for Firmware Images. In *CCS* (Vienna, Austria). Association for Computing Machinery, New York, NY, USA, 480–491. https://doi.org/10.1145/2976749.2978370

[20] Artyom V. Gorchakov, Liliya A. Demidova, and Peter N. Sovietov. 2023. Analysis of Program Representations Based on Abstract Syntax Trees and Higher-Order Markov Chains for Source Code Classification Task. *Future Internet* 15, 9 (2023). https://doi.org/10.3390/fi15090314

[21] Irfan Ul Haq and Juan Caballero. 2021. A Survey of Binary Code Similarity. *ACM Comput. Surv.* 54, 3, Article 51 (apr 2021), 38 pages. https://doi.org/10.1145/3446371

[22] James W. Hunt and Thomas G. Szymanski. 1977. A Fast Algorithm for Computing Longest Common Subsequences. *Commun. ACM* 20, 5 (may 1977), 350–353. https://doi.org/10.1145/359581.359603

[23] Petr Jancar. 2014. Equivalences of Pushdown Systems Are Hard. In *FOSSACS (Lecture Notes in Computer Science, Vol. 8412)*, Anca Muscholl (Ed.). Springer, 1–28. https://doi.org/10.1007/978-3-642-54830-7_1

[24] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM – software protection for the masses. In *SPRO*. IEEE, Washington, DC, US, 3–9.

[25] Song Liu. [n. d.]. Awesome Binary Similarity – The online github collection. https://github.com/SystemSecurityStorm/Awesome-Binary-Similarity. [Online; accessed 29-Jan-2024].

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI* (Chicago, IL, USA). Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[27] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. AAAI Press, Palo Alto, CA, US, 1287–1293.

[28] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 89–100. https://doi.org/10.1145/1250734.1250746

[29] Marcelo Nogueira and Sérgio Medeiros. 2024. Classifying C++ Solutions Based on Their Energy Profile. In *SBLP* (Curitiba/PR). SBC, Porto Alegre, RS, Brasil, 98–101. https://sol.sbc.org.br/index.php/sblp/article/view/30263

[30] Dinglan Peng, Shuxin Zheng, Yatao Li, Guolin Ke, Di He, and Tie-Yan Liu. 2021. How could Neural Networks understand Programs?. In *ICML (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, Online, 8476–8486. http://proceedings.mlr.press/v139/peng21b.html

[31] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 709–724. https://doi.org/10.1109/SP.2015.49

[32] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

[33] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the Hidden Power of Compiler Optimization on Binary Code

Difference: An Empirical Study. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035

[34] H. G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. http://www.jstor.org/stable/1990888

[35] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. 2021. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.* 51, 2 (2021), 353–384. https://doi.org/10.1002/spe.2907

[36] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (2020), 27 pages. https://doi.org/10.1145/3418463

[37] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul PS, Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. 2024. VEXIR2Vec: An Architecture-Neutral Embedding Framework for Binary Similarity. arXiv:2312.00507 [cs.PL] https://arxiv.org/abs/2312.00507

[38] Zheng Wang, Ken Pierce, and Scott McFarling. 2000. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism* 2 (2000), 1–20.

[39] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2023. Challenging Machine Learning-Based Clone Detectors via Semantic-Preserving Code Transformations. *IEEE Trans. Softw. Eng.* 49, 5 (may 2023), 3052–3070. https://doi.org/10.1109/TSE.2023.3240118

[40] Anshunkang Zhou, Yikun Hu, Xiangzhe Xu, and Charles Zhang. 2024. ARCTURUS: Full Coverage Binary Similarity Analysis with Reachability-Guided Emulation. *ACM Trans. Softw. Eng. Methodol.* (jan 2024). https://doi.org/10.1145/3640337 Just Accepted.

Received 2024-11-01; accepted 2024-12-21