A Game-Based Framework to Compare Program Classifiers and Evaders

Thaís Damásio
UFMG
Minas Gerais, Brazil
thais.damasio@dcc.ufmg.br

Michael Canesche
UFMG
Minas Gerais, Brazil
michaelcanesche@dcc.ufmg.br

Vinícius Pacheco
UFMG
Minas Gerais, Brazil
vinicius.pacheco@dcc.ufmg.br

Marcus Botacin
Texas A&M University
Texas, USA
botacin@tamu.edu

Anderson Faustino da Silva
UEM
Paraná, Brazil
anderson@din.uem.br

Fernando M. Quintão Pereira
UFMG
Minas Gerais, Brazil
fernando@dcc.ufmg.br

Abstract

Algorithm classification consists in determining which algorithm a program implements, given a finite set of candidates. Classifiers are used in applications such malware identification and plagiarism detection. There exist many ways to implement classifiers. There are also many ways to implement evaders to deceive the classifiers. This paper analyzes the state-of-the-art classification and evasion techniques. To organize this analysis, this paper brings forward a system of four games that matches classifiers and evaders. Games vary according to the amount of information that is given to each player. This setup lets us analyze a space formed by the combination of nine program encodings; seven obfuscation passes; and six stochastic classification models. Observations from this study include: (i) we could not measure substantial advantages of recent vector-based program representations over simple histograms of opcodes; (ii) deep neural networks recently proposed for program classification are no better than random forests; (iii) program optimizations are almost as effective as classic obfuscation techniques to evade classifiers; (iv) off-the-shelf code optimizations can completely remove the evasion power of naïve obfuscators; (v) control-flow flattening and bogus-control flow tend to resist the normalizing power of code optimizations.

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

Keywords: algorithm classification, obfuscation

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

ACM ISBN 979-8-4007-0101-6/23/02…$15.00
https://doi.org/10.1145/3579990.3580012

ACM Reference Format:
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 (CGO '23), February 25 – March 1, 2023, Montréal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3579990.3580012

1 Introduction

The problem of *algorithm classification* can be informally defined as follows: given a finite set of different specifications of algorithms, plus a program that implements one of them, find which algorithm the problem implements. This challenge has been called *task classification* by Allamanis et al. [1] and *program classification* by Mou et al. [27]. The name *algorithm classification* seems to have its origin in Ben-Nun et al. [4]'s work. The problem of algorithm classification has been much studied in the machine learning literature—for an overview, we recommend Section 2 of Peng et al. [32]. Algorithm classification is important because it has many applications: redundancy elimination [3] and name replacement [2, 21] in source code, malware identification [22], plagiarism detection [6, 29], task identification [42], etc.

Classifiers: the Good Side. As consequence of Rice [35]'s Theorem, constructing a perfect algorithm classifier is impossible. Thus, typical solutions to this problem are of stochastic nature. Given the importance of the problem, and its open essence, in recent years, many approaches have been proposed for the construction of classifiers. These approaches vary in terms of the way to represent programs, or in the classification model adopted, or in the dataset used to train the classifier. Program representations include, for example, Inst2Vec [4], Code2Vec [2], Ir2Vec [40], Asm2Vec [16], Milepost [28] and ProGraML [8]. Classification models include neural networks proposed by Mou et al. [27], Cummins et al. [9] and Brauckmann et al. [5], for instance. And large datasets of reference algorithms have been released by Mou et al. [27], and Puri et al. [33], for instance.

Evaders: the Bad Side. Just like there exists much effort to improve the precision of algorithm classifiers, there exists also effort to evade them. Evasion enhances the ability of malware to escape anti-virus systems [44] or improves the chances that plagiarism might rest undiscovered [15]. The common approach to defeat algorithm classifiers is code transformation [36]. Typically, evaders transform programs via code obfuscation; however, recent studies have demonstrated that standard compiler optimizations are also effective to hide programs [34]. Given that there are so many evasion techniques, and yet so many ways to implement classifiers, an immediate question that we pose, as a research community, is where do we stand in this arms race. The goal of this paper is to provide some answers to this question.

The Contributions of This Work. This paper proposes a system of four games to compare algorithm classifiers ("classifiers" for short) and evaders. Every game uses a variation of Mou et al. [27]'s balanced data set of solutions to programming problems. The classifier is trained with part of this dataset, and the evader challenges it with the rest. As we explain in Section 2, games differ on the resources given to each player. Two games are symmetric: the classifier knows any obfuscation strategy that is allowed to the evader. The two other games are asymmetric: the evader can transform programs with an unknown obfuscator. In one of these games, the classifier can try to revert the effects of obfuscation using code optimizations as a normalization strategy.

The Arena. This paper does not propose classification approaches nor obfuscation strategies. Instead, as we explain in Section 3, we create a playing arena by extracting artifacts from previous work. In common these artifacts have the fact that they manipulate programs in the LLVM [23] intermediate representation (IR). We have built classifiers by combining nine different program embeddings (i.e., a vector-based representation of a program) with six different stochastic classification models. We have built evaders out of eight code obfuscation techniques. Section 4 compares classifiers and evaders using a curated version of Mou et al. [27]'s Poj-104 dataset, plus 48 implementations of the Mirai malware [19]. From this comparison, we draw a number of conclusions, some of which we list below.

- **Histograms:** When evaluating symmetric games, we observed no advantage of program embeddings designed for code classification over histograms of opcodes. Although simple, histograms resist well against transformations such as control-flow flattening, and can work in tandem with different classification models.
- **Models:** we could observe no improvement of neural networks previously used for program classification [45] over an off-the-shelf implementation of random forests. Random forests seem to surpass Zhang et al.'s neural networks by a wider margin once evaders are allowed to transform programs.
- **Optimizers:** optimizers are as effective as code obfuscators as an evasion strategy, while producing code about 60x faster. However, code optimization tends to resist poorly to a classifier that is aware of the optimization approach used by the evader, whereas there exist obfuscation techniques that can resist disclosure.
- **Normalization:** code optimizations revert the effects of many obfuscation strategies. A classifier that is allowed to optimize programs with `clang-O3` is impervious to transformations like those proposed by Zhang et al. [46] (built after Devore-McDonald and Berger [15]'s). However, obfuscation approaches like control-flow flattening and bogus control flow tend to resist optimization-based normalization.

2 The Game Framework

Definition 2.1 formalizes the notion of *programming problem*. Definition 2.1 leaves the concept of *reference function* vague on purpose: it could be a black-box implementation of a program; a table with inputs and expected outputs; a specification written in natural language, etc. Example 2.2 shows examples of typical programming problems.

Definition 2.1 (Programming Problem)**.** Let f_{ref} be a *reference oracle*: a function whose implementation is unknown, but that can be consulted (i.e., invoked). Function f_{ref} defines a *programming problem*. A solution to this problem is a function s of known implementation that produces the same output as f_{ref}, when given the same inputs.

Example 2.2. Poj-104 [27] is a set of 104 problems taken from an online programming judge. Each problem consists of 500 sample solutions, each written by a potentially different human programmer. Problems are defined by a set of inputs and the expected outputs. CodeNet [33] is a set of 4,053 programming problems with 13,916,868 submissions written in 55 different programming languages, out of which 53.6% represent correct solutions to the reference oracles. These oracles are tables with inputs and expected outputs.

Definition 2.3 formalizes *algorithm classification* as the challenge of guessing which problem a program solves. Notice that algorithm classification deals with a closed universe of algorithms: it involves a finite set of possible problems from which the classifier must choose one. This restriction is in contrast to the problem of *code summarization* [38], which assumes an open universe of possible algorithms.

Definition 2.3 (Algorithm Classification)**.** Given a set $\{f_1, f_2, \ldots f_m\}$ of m *different* programming problems, plus a solution s for one of them, the algorithm classification problem asks to find the problem f_i, $1 \leq i \leq m$ that s solves.

A *classifier* categorizes programs into algorithms. A program that transforms other programs to evade classification is an *evader*. In this paper, transformations are either *optimizations*, which try to make a program more efficient; or *obfuscations*, which try to hide the purpose of the program, often making it less efficient. From these concepts, we define an *adversarial game* as follows:

Definition 2.4 (Adversarial Game)**.** A game consists of a *classifier* C, an *evader* E, a set F of m programming problems $\{f_1, f_2, \ldots, f_m\}$, a set S of n challenges $\{s_1, s_2, \ldots, s_n\}$ and an *accuracy threshold* $K, 0 \leq K \leq 1$. Each challenge $s_i \in S, 1 \leq i \leq n$ must solve exactly one problem $f_j \in F, 1 \leq j \leq m$. For each s_i, the game proceeds as follows:

1. The evader produces a solution $s'_i = E(s_i)$;
2. The classifier chooses problem $f_j = C(s'_i), 1 \leq j \leq m$.

The classifier wins the game if it can find the right problem f_j that the solution s'_i solves with probability greater than K. It loses the game otherwise.

The evader, according to Definition 2.4, is allowed to modify a program before giving it to the classifier. Modifications must preserve semantics. Thus, if s is a solution to a programming problem f (as per Definition 2.1), and E is an evader, then $s' = E(s)$ is also a solution to f. From this observation, Figure 1 lists the four games that we consider in this paper.

Game 0 (symmetric)	
Classifier	0.8 Dataset
Evader	0.2 Dataset

Game 1 (asymmetric)	
Classifier	0.8 Dataset
Evader	0.2 Dataset + Transformer

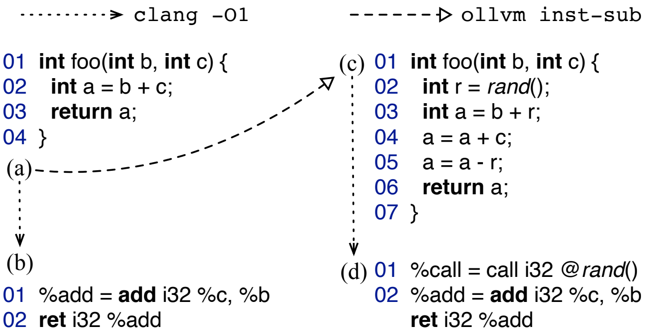
Game 2 (symmetric)	
Classifier	0.8 Dataset + Transformer
Evader	0.2 Dataset + Transformer

Game 3 (asymmetric)	
Classifier	0.8 Dataset + Optimizer
Evader	0.2 Dataset + Transformer

Figure 1. The four games to evaluate classifiers and evaders.

Symmetric vs Asymmetric Games. In Figure 1, a "symmetric" game means that classifiers and evaders have access to the same resources. An "asymmetric" game means that classifiers and evaders have access to different resources. In **Game0**, programs are used without transformation, e.g., the evader E is the identity function. In **Game1**, the evader is allowed to transform the challenge that it gives to the classifier, but the classifier is tuned with the original training set. In **Game2**, classifier and evader have access to the same code transformation, which can be either an obfuscation or an optimization. The transformation is one-way: if $s' = E(s)$, the classifier cannot recover s by inspecting s'. However, it can apply the transformation onto the training set. In **Game3**, the two players have access to some code transformation, albeit not the same. Our hypothesis is that optimizing transformations let the classifier normalize programs, approximating challenges to the training set. Example 2.5 backs up this hypothesis with some intuition.

Example 2.5. Figure 2 illustrates **Game3**. The evader modifies a program before challenging the classifier. In this example, the evader uses O-LLVM's "instruction substitution" [20]. This pass replaces arithmetic instructions with other instructions that implement the same semantics. The classifier, in turn, can optimize programs in its training set, using, for instance, the `-O1` level of `clang`. When given a challenge, like function `foo`, the classifier applies the same transformation, e.g., `clang -O1`, onto `foo`, before classifying it. In this example, `clang -O1` partially undoes the transformations carried out by the evader. Notice that the example uses source code for clarity, but the transformations that we evaluate in this paper happen in the LLVM intermediate representation.



```
int foo(int b, int c) {
  int a = b + c;
  return a;
}
```

```
%add = add i32 %c, %b
ret i32 %add
```

```
int foo(int b, int c) {
  int r = rand();
  int a = b + r;
  a = a + c;
  a = a - r;
  return a;
}
```

```
%call = call i32 @rand()
%add = add i32 %c, %b
ret i32 %add
```

Figure 2. (a) A solution to the programming problem "sum up two integers". (b) The optimized LLVM representation of function `foo`. (c) A version of `foo` obfuscated with O-LLVM's instruction substitution. (d) The version of the obfuscated function optimized with `clang -O1`.

3 The Classification Arena

This paper reuses artifacts from previous works to build classifiers and evaders. These artifacts work on programs in the LLVM intermediate representation; hence, support some level of cross-language analysis [30]. However, our experiments use only codes derived from C and C++ programs. Figure 3 shows the two embeddings, nine models and six code normalizers used to build classifiers. In total, this paper evaluates 2×9×6 classifiers. Figure 4 shows the nine evaders tried in this paper. Evaders differ on the transformation that they apply on the challenge, before giving it to the classifier. **Game0** uses a passive evader, which does not use any transformation. Figure 4 places this player in the last row. The rest of this section explains the embeddings, models and transformations that we have used in this paper.

3.1 Program Embeddings

A program is a string of characters. This string needs to be transformed into a numerical format to be classified. Definition 3.1 calls this format a *program embedding*.

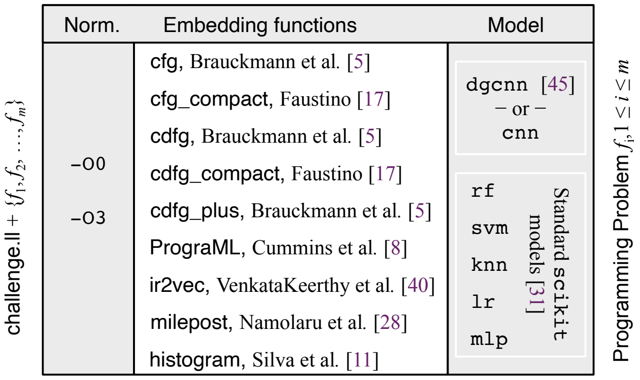

Norm.
Embedding functions
Model
challenge.ll + $\{f_1, f_2, \ldots, f_m\}$
-O0
-O3
cfg, Brauckmann et al. [5]
cfg_compact, Faustino [17]
cdfg, Brauckmann et al. [5]
cdfg_compact, Faustino [17]
cdfg_plus, Brauckmann et al. [5]
ProGraML, Cummins et al. [8]
ir2vec, VenkataKeerthy et al. [40]
milepost, Namolaru et al. [28]
histogram, Silva et al. [11]
dgcnn [45]
– or –
cnn
rf
svm
knn
lr
mlp
Standard scikit models [31]
Programming Problem f_i, $1 \leq i \leq m$

Figure 3. The different classifiers used in this work. "Norm." denotes *code normalizer*, i.e., a transformation that the classifier is allowed to apply on code, to ease the task of doing algorithm classification in **Game3**. We have experimented with `clang -O3` and with `clang -O0` (no normalization).

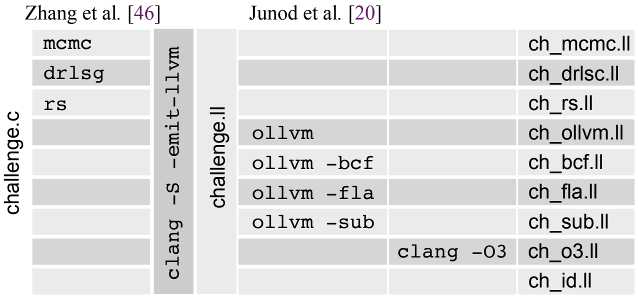


Figure 4. The nine evaders evaluated in this paper.

Definition 3.1 (Program Embedding)**.** A *program embedding* is an array of numerical values, i.e., a tensor, that represents a program. The function that maps programs onto embeddings is called an *embedding function.*

This paper evaluates the embedding functions in Figure 3. Our criteria to choose embeddings was (i) the existence of a publicly available artifact and (ii) the input of said artifact being the LLVM intermediate representation (IR)[1]. Ir2Vec, Milepost and Histogram map programs in the LLVM IR into arrays. The other embeddings transform a graph-based representation of the program into a tensor formed by three arrays representing vertex attributes, edge attributes and adjacencies. Figure 7 of Brauckmann et al. [5] shows how a graph can be converted into an array. Graph-based formats are extracted from programs in the LLVM IR. These graphs differ on the amount of information that they expose: instructions, control flow, data dependencies, function calls, etc. Two graph representations are compact (cfg_c and cdfg_c) [17], meaning that instructions are grouped into basic blocks, instead of represented as individual nodes.

3.2 Classification Models

Classifiers use stochastic models to solve algorithm classification. We define a *model* as follows:

Definition 3.2 (Model)**.** A *model* is a function that receives (i) a program embedding; (ii) a set F of m programming problems; and (iii) a challenge s, and outputs the problem $f_i \in F$ that s is believed to solve.

This paper evaluates the six models in Figure 3. Five of these models are standard implementations available in `SciKit-Learn` [31]: `rf`: random forest; `svm`: support vector machine; `knn`: k-nearest neighbors; `lr`: logistic regression; and `mlp`: multi-layer perceptron. The sixth model (`dgcnn`) is the Deep Graph Convolutional Neural Network proposed by Zhang et al. [45]. This model—a learning architecture that receives attributes characterizing graphs—is built as follows:

1. four graph convolutional layers, with 32, 32, 32, and one unit with hyperbolic tangent activation;
2. a one-dimensional convolutional layer;
3. a max pooling layer;
4. a one-dimensional convolutional layer;
5. a dense layer followed by a dropout layer;
6. a final dense layer that does the classification.

Zhang et al.'s model will only be used as is in Section 4.1 of this paper, because it requires graph-based program embeddings. Thus, `dgcnn` is used with the six graph-based embeddings from Figure 3. When applied onto the three other embeddings: Ir2Vec, Milepost and Histogram, we use a simpler version of it, `cnn`, which lacks the four first layers. These layers only exist to merge the attributes of vertices and edges into a single array, which is then used to start the classification process. Thus, when receiving arrays as the program embedding, these layers find no service. For comparison purposes, we shall evaluate a standard implementation of a neural network from `SciKit` (the `mlp` model), with only one hidden layer with 100 perceptrons and ReLu activation—which uses between 10 and 100x less memory.

3.3 Code Transformations

Evaders transform programs to confuse classifiers. This paper recognizes nine different transformations, which Figure 4 shows. One of these transformations is the identity function; the other is the sequence of compiler optimizations applied by `clang-O3`. Three other transformations, `bcf`, `fla` and `sub`, are available in `O-LLVM` [20], a code obfuscation framework built on top of LLVM. These transformations can be used in combination or separately. In this paper we either use them all together (in the pass that we call simply `ollvm`), or use them individually. The three transformations are as follows:

- **bcf:** *Bogus Control Flow* inserts extra jumps into the program's control flow graph. These jumps are controlled by conditions that are always false, but that are not resolved by LLVM's standard optimizations.

[1] We tried to use Ben-Nun et al. [4]'s Inst2Vec, but failed to reproduce their experiments: the artifact runs out of memory even for small training sets.

- **fla:** *Control-Flow Flattening* replaces every jump in the program's CFG with a case from a switch statement within a loop. A counter at the end of every basic block determines the next block to be executed.
- **sub:** *Instruction Substitution* replaces logic and arithmetic instructions with sequences of commands that are semantically equivalent (See Example 2.5).

Three other obfuscations used in this paper were proposed by Zhang et al. [46] to evade stochastic clone detectors. These obfuscation techniques operate directly in the source code of programs. Each of them is a different strategy to combine 15 simpler transformations. These simpler transformations consist, for instance, in replacing a `for` loop with a `while` loop; replacing a `switch` statement with a chain of conditional branches; replacing constants with arithmetic operations; etc. The three strategies that we reuse from Zhang et al. are:

- **rs:** combines the 15 transformations randomly, but without repetition, via a *Random Search.*
- **mcmc:** uses a *Markov-Chain Monte Carlo* algorithm to combine transformations, favoring sequences that lead to programs harder to understand.
- **drlsg:** uses a *Deep Reinforcement Learning* to combine transformations, maximizing the distance between the new program and its original version.

We could not make Zhang et al.'s scripts work directly. Thus, we use its dataset: Zhang et al.'s repository contains transformed versions of all the programs in Poj-104. We had to fix some of these programs, inserting header files, to ensure that they could be all compiled via `clang`.

4 Evaluation

This section provides answers to eight research questions, discussed in Sections 4.1-4.8. The hardware adopted in these experiments bears no influence on their results, except in Section 4.6. Experiments were carried out on Linux Ubuntu 20.4. The intermediate representation of programs was extracted with LLVM 4.0, except in Section 4.1. In that case, we use LLVM 15.0. We adopt an old version of LLVM, because that is the only version which can be used with `O-LLVM`. Section 4.1 uses LLVM 15.0 because that is the only version that provides passes to extract all the program representations. Box plots, whenever used, summarize ten samples.

Datasets: Sections 4.1, 4.2, 4.3, 4.4 and 4.7 use a perfectly balanced dataset formed by the 104 classes of programming problems released by Mou et al. [27]. Each class contains 500 solutions to a programming problem; hence, in total, we use 104 × 500 programs (Section 4.1 uses a subset of this dataset). Solutions were presumably written by different people. Classifiers are trained with a random selection of 375 samples from each class (the *training set*), and challenged with 125 samples (the *test set*). Training and classification happens anew in Sections 4.1, 4.2, 4.3, 4.4 and 4.7; thus, results might vary slightly, even if they discuss the same experiment.

Evaluation Metric: our evaluations of classification games use *perfectly balanced* datasets, meaning that the number of programming problems used to train the classifiers, and the number of challenges (s_j in Def. 2.4) are equally distributed among the different classes of problems. In this case, the F1-score and the simple Accuracy (number of hits divided by number of tries) yield the same information (as an illustration, we report both metrics in Figure 12). Therefore, we shall report only accuracy in most of our analyses.

4.1 RQ1: Comparing Program Embeddings

Question 1 (RQ1). *How effective are typical program embeddings to support algorithm classification?*

This section compares the embeddings in Figure 3 with 32 classes of programming problems taken randomly from Poj-104. We do not use all the 104 classes of problems in Poj-104 to reduce evaluation times. In total, running one round of evaluation (nine representations) takes 11-12 hours in a commodity machine (2.0GHz); and each experiment requires ten rounds. The comparison uses Zhang et al. [45] neural network—the only model that fits all the embeddings. The other models can only receive histograms.

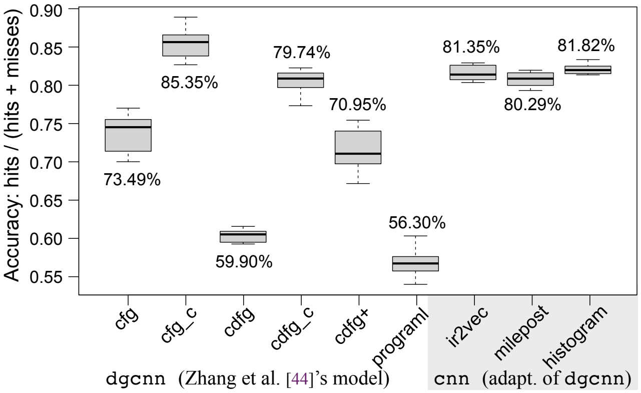


Figure 5. Comparison of program embeddings in **Game0** using 32 problems. Numbers next to boxes are means.

Discussion: Figure 5 compares nine embeddings in **Game0**. The most accurate results were observed when CFG_C was the embedding used to classify programs. Mean accuracy was 85.36%. We could not observe statistically significant differences (pairwise p-values above 0.2) between CDFG_C, IR2VEC, MILE- POST and HISTOGRAM: they all had mean accuracy between 81 and 82%. We found this result surprising, because CFG_C also uses histograms: it produces a histogram of opcodes for every basic block in the program, plus an adjacency matrix to represent control-flow information. And yet, it is only marginally better than HISTOGRAM, which is a vector of 63 positions counting instruction opcodes. In other words, CFG_C is asymptotically more costly than HISTOGRAM, but its observed benefit is small.

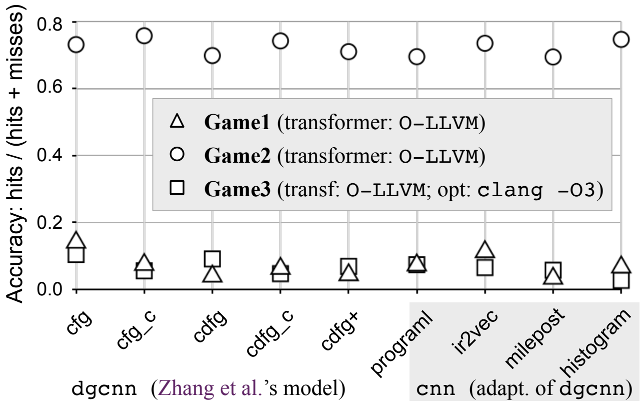


Figure 6. Comparison of program embeddings in **Games 1**, **2** and **3** using 32 problems. Dots are averages of 10 samples.

Trends observed in Figure 5 emerged in the other games, as Figure 6 shows. In this case, Histogram and cfg_c yield the best accuracies for **Game2**: about 76%. Nevertheless, the different embeddings produce similar results. Accuracy drops on the asymmetric games. This result is the consequence of the obfuscator used: O-LLVM with its four transformation passes. The next sections will discuss this result in more detail. The cdfg+ embedding seems to better resist code transformations, but its accuracy was never above 20%.

4.2 RQ2: Comparing Classification Models

Question 2 (RQ2). *How effective are typical classification models to support algorithm classification?*

This section compares the accuracy and the memory consumption of different models in regards to **Game0**. Results are shown as boxplots aggregating ten executions of each model. Variations, albeit small, are possible, because the six different models that we compare, except knn, have parameters that are initialized with random weights.

Discussion: Figure 7 shows the relative performance of the six models on the test set formed by 104 problems (f_i, $1 \leq i \leq 104$); each with 125 challenges (s_j, $1 \leq j \leq 125$). Random forest showed the best accuracy (80.0%). This number means that, given a program and 104 problems that it might solve, rf is correct in four out of five guesses. However, the difference to Zhang et al. [45]'s neural network or to SciKit's is very small: less than 1.0%. Nevertheless, the difference is still statistically significant within a confidence level of 0.99. Although the difference between Zhang et al.'s cnn and SciKit's mlp is small, the latter uses much less memory: mlp (also knn, svm and lr) runs with less than 0.5GB of RAM. Cnn uses 2.0GB, and rf uses 2.2GB.

4.3 RQ3: Measuring Evasion

This section evaluates **Game1** and **Game2** (see Figure 1) using the six models analyzed in Section 4.2. We consider only the Histogram embedding, as it is the only one that can be passed as input to the different models.

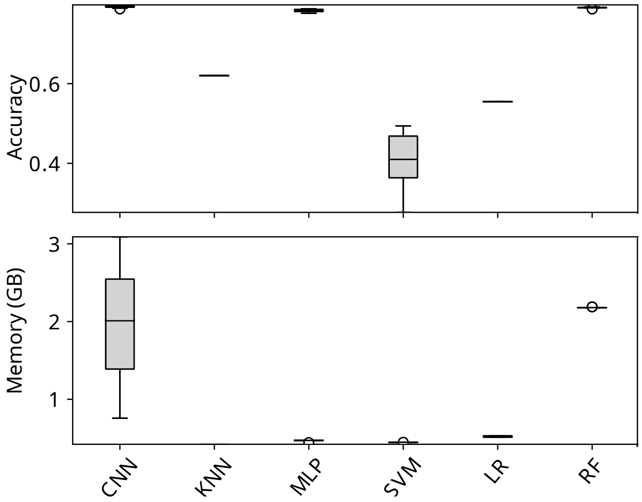


Figure 7. Comparison of different models used in **Game0**, using 104 problems and the Histogram embedding.

Question 3 (RQ3). *How effective are typical code obfuscation techniques as a means to evade algorithm classification?*

Figure 8 evaluates **Game1**. Each dot is the average of ten measurements. The first column of Figure 8 shows the baseline; i.e., the average of the accuracy results earlier seen in Figure 7. Almost every obfuscation strategy reduces the accuracy of the classifier. **Drlsg**, which Zhang et al. [46] have introduced as the most effective obfuscation approach among their choices (including **mcmc** and **rs**) has no effect upon the classifier. Once we inspect the bytecodes that **drlsg** produces, the SSA conversion that LLVM uses reverts all the effects of it. Control-flow flattening and instruction substitution have almost no effect on the classifier that uses random forests. Flattening barely changes the histogram of instructions in a program—an observation that advocates for the usage of histograms of opcodes as a classification embedding. Substitution changes the instructions, but preserves proportions; hence, random forests are still able to perform effective classification.

Figure 9 evaluates **Game2**. In this case, the classifier is aware of the obfuscation strategy adopted by the evader. Thus, each sample in the training dataset is obfuscated before being given to the classifier. Notice that the classifier is trained only with obfuscated codes, not with original programs. The adversary, similarly, only uses obfuscated challenges. In this case, classifiers and evaders behave almost like in **Game0**. Thus, knowledge of the obfuscation approach is enough to give the classifier power to resist evasion.

To explain the results seen in Figures 8 and 9, we compare the average distance between histograms of original and obfuscated programs. We use Euclidean Distance computed in

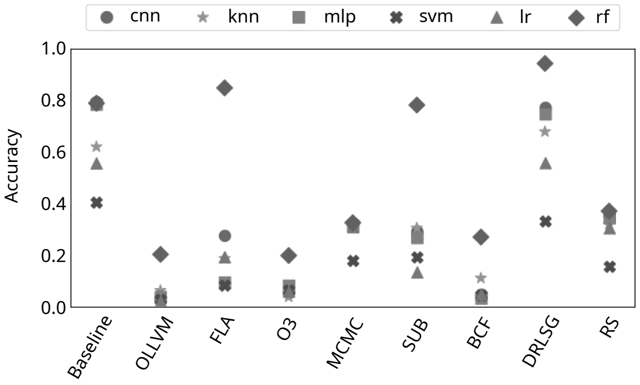


Figure 8. Evaluation of **Game1** on 104 problems, using the Histogram embedding plus six different models.

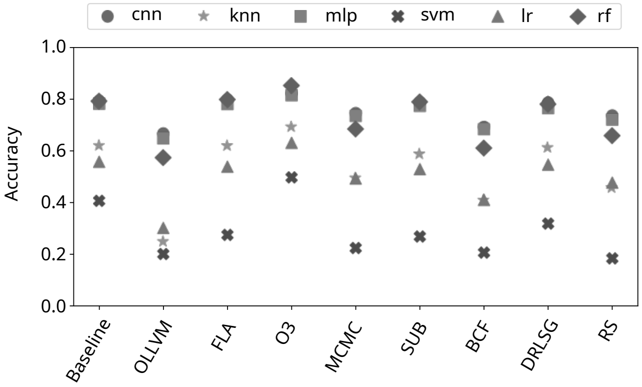


Figure 9. Evaluation of **Game2** on 104 problems, using the Histogram embedding plus six different models.

a space of 63 dimensions (the number of opcodes). Figure 10 shows this comparison. The greater the distance between old and new histograms, the greater the capacity of an evader to deceive the classifier. In this sense, the most effective evasion techniques are the optimizations used by `clang -O3` and the combination of transformations used by `O-LLVM`.

4.4 RQ4: The Normalization Hypothesis

Question 4 (RQ4)**.** *Can compiler optimizations improve algorithm classification by reverting the effects of obfuscation?*

This section evaluates **Game3** to verify if optimizations can revert the effects of obfuscation techniques. To this end, we train the classifiers with normalized programs (the classifier is not aware of any obfuscator the evader can use). When receiving a challenge—which consists of an obfuscated program—the classifier optimizes that code.

Discussion: Figure 11 summarizes the evaluation of **Game3**. `O-LLVM` is resistant against code optimizations. However, this ability is due to bogus control flow (`bcf`)—which cannot be easily optimized—plus an interesting accident: the application of optimizations onto code subject to control-flow flattening changes substantially the mix of instructions in that code (important: the classifier has no access to flattened samples). In this case, optimizations caused the opposite of what we expected: they increase the ability of `fla` to hide the purpose of a program. And yet, when used alone, `fla` fails to deceive a histogram-based classifier (see Fig. 8). The obfuscation techniques designed by Zhang et al. [46] are ineffective in face of optimizations. These transformations might confuse source-code analyses; however, their effects disappear from optimized codes.

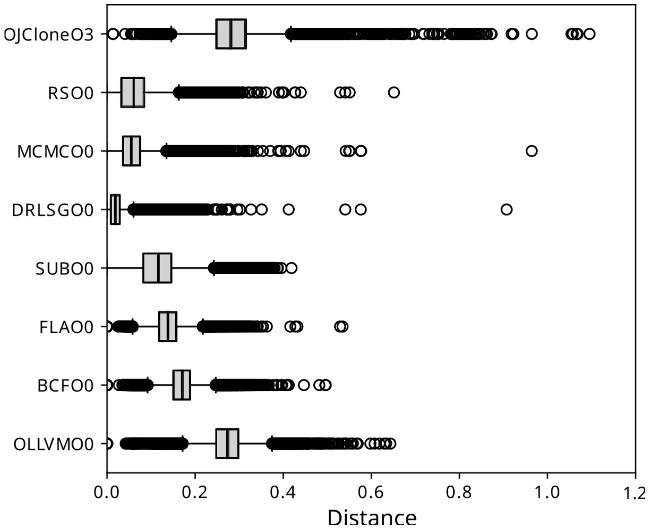


Figure 10. Analysis of the distance between program embeddings extracted from different obfuscated codes.

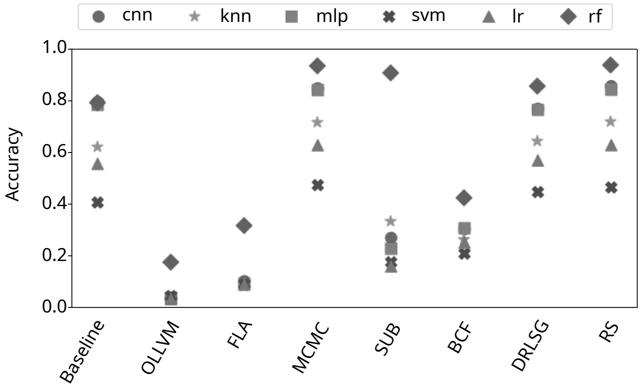


Figure 11. Evaluation of **Game3** with the Histogram embedding plus six models. The normalizer is `clang -O3`.

4.5 RQ5: On the Number of Available Classes

Question 5 (RQ5)**.** *How does the effectiveness of algorithm classifiers vary with the number of classes of algorithms?*

Classification becomes harder as the number of classes grow. For instance, a random classifier has an expected hit rate of 50% when dealing with two classes, and of 0.96% when dealing with the 104 classes from Poj-104. Our classifiers enjoy higher accuracy. This section investigates how this accuracy varies with the number of classes of algorithms.

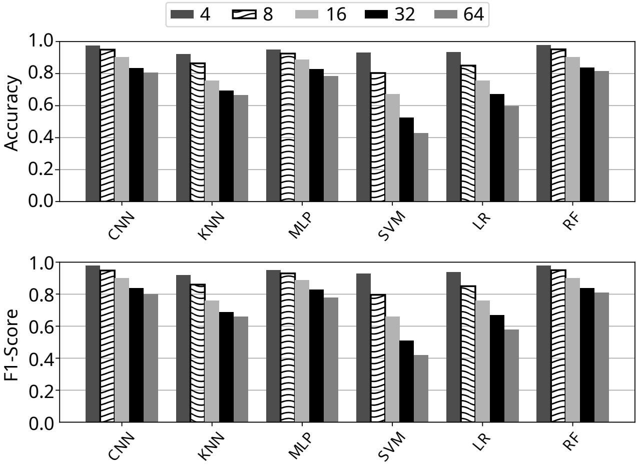


Figure 12. Evaluation of **Game0** using the Histogram embedding, considering a varying number *m* of classes of programming problems.

Discussion: Figure 12 shows the accuracy (and the F1-score) of the histogram-based classifiers in **Game0**, when dealing with $2^i, 2 \leq i \leq 6$ classes of programming problems. Figure 7 already shows accuracy for the maximum number of classes: 104. As expected, accuracy drops almost linearly with the number of classes; however, the constant of the linear factor is small. The classifier built with random forests still shows a hit rate of about 80% when dealing with 64 classes—more than 50x the expected accuracy of a random classifier.

4.6 RQ6: Performance Impact

Question 6 (RQ6). *How does the performance of code produced by typical obfuscators and typical optimizers vary?*

We evaluate Question 6 on the C programs from "The Benchmark Game" (https://benchmarksgame-team.pages.debian.net/benchmarksgame/). We show numbers for `clang -O3` and `O- LLVM`. Numbers for Zhang et al. [46]'s transformations cannot be shown, for they have released only obfuscated versions of Poj-104, but not the transformers.

Discussion: Figure 13 compares the running times of codes produced with `clang-O3` and with `O-LLVM`. Times are relative to programs compiled with `clang -O0`. `O-LLVM` slows down every program; `clang-O3` speeds up all of them. The 16 programs, compiled with `clang -O0`, run in 203.45 seconds (std = 3.08 sec). On average (geometric mean), the 16 obfuscated programs are 8.33x slower than the programs compiled with `clang -O0`. The optimized programs are 2.32x faster than what `clang -O0` produces. Yet, variations are important: obfuscated `ary3` experiences a slowdown of more than 30x, and optimized `matrix` [multiplication] experiences a speedup of almost 7x. Optimized programs were 1.23x-136.18x faster than their obfuscated counterparts.

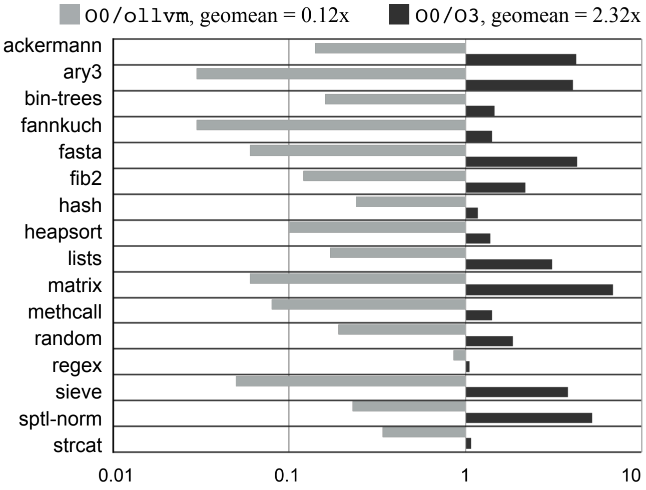


Figure 13. Running times of optimized and obfuscated programs. Times are relative to code produced by `clang -O0`.

4.7 RQ7: Detecting the Obfuscator

Question 7 (RQ7). *Can the algorithm classifiers evaluated in this paper detect the obfuscator used in games 1 and 2?*

Game2 relies on the premise that the classifier "knows" the obfuscation strategy used by the evader. However, discovering the obfuscation strategy applied onto programs is not trivial, as this section shows.

Experimental Setup. To answer Question 7, we have evaluated a classifier on a game formed by a set of 10 code transformers, plus a dataset of 500 programs. We train the classifier onto 10 × 400 transformed programs, and challenge it with the 10 × 100 remaining programs. We consider the following transformers: (i): `clang -O0`; (ii): `clang -mem2reg`; (iii): `clang -O3`; (iv): `ollvm -bcf`; (v): `ollvm -fla`; (vi): `ollvm -sub`; (vii): `drlsc`; (viii): `mcmc`; (ix): `rs` and (x): `ga`. We thus have ten classes to classify programs. Class-ii is formed by programs in the LLVM IR optimized with just the `mem2reg` pass, which maps variables onto symbolic registers, instead of leaving them in memory. Class-x is the genetic algorithm used by Zhang et al. [46]. We could not use it in the other experiments, because we could not apply it onto every program in Poj-104. However, `ga` was available to answer Question 7, given the restricted datasets used in this section. We evaluate Question 7 onto four datasets, each containing 5,000 samples: `dataset1`: a transformed version of each one of the 500 programs from a single—random—problem taken from Poj-104; `dataset2`: a transformed version of each one of the 5 random solutions of a different problem from Poj-104 (we eliminated four classes randomly, to remain with 100 classes); `dataset3`: same as `dataset1`, except that each transformer is given 500 solutions from the same programming problem. `dataset4`: same as `dataset2`, except that each transformer is given 5 solutions from the same programming problem. Notice `dataset1` and `dataset2` give the same programs to each transformer, whereas the other two datasets give different programs to each transformer.

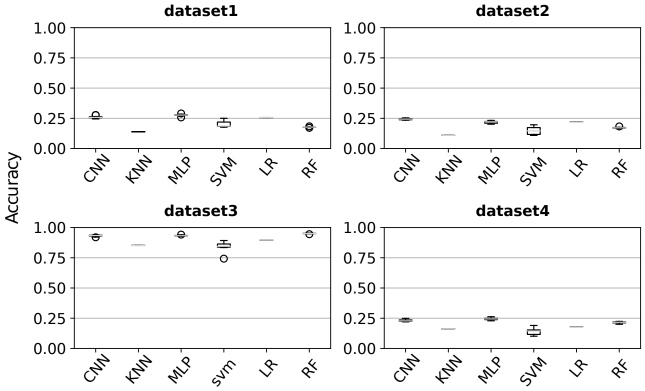


Figure 14. Evaluation of Question 7 on four different datasets. Programs in each dataset are originally produced with `clang -O0`, and then transformed via either optimizations or obfuscations.

Discussion: Figure 14 compares the performance of the classifier on the four different setups that we consider. Notice that in three setups the performance of the classifier is approximately the same: it obtains a hit rate of 25%. Although higher than a random hit rate (expected 10%), this result shows that "obfuscation classification" is harder than algorithm classification—at least given the stochastic tools evaluated in this paper. The only situation where we observed a high hit rate was in `dataset3`. However, this is a spurious correlation: every obfuscator is applied onto a different programming problem. In this case, the classifier is rather discovering that programming problem, not the obfuscator used to transform the program.

4.8 RQ8: Actual Malware

Question 8 (RQ8). *Can the algorithm classifiers evaluated in this paper be used to identify real-world malware?*

This section evaluates the classifiers discussed in Section 3 on 48 versions of Mirai[2], a malware that turns networked devices running Linux into remotely controlled bots.

Building Mirai Identifiers. A classifier cannot be effectively trained with 48 samples. Thus, to increase the number of samples, we proceed as follows. First, we separate 36 *positive* samples (i.e., actual malware) for training; and add to this suite 36 *negative* samples (i.e., benign software) taken from SPEC CPU2017. Let this initial collection be the *seed suite*. The benign samples were chosen by size: for each negative sample, we chose the C file from SPEC that had the closest size, in the number of LLVM instructions. From this seed collection, we can build seven classifiers by choosing a classification model (see Figure 3) and varying the training set. Each training set t_i, $1 \leq i \leq 6$, is formed by adding to t_{i+1} all the programs in t_i, plus a *transformed version* of the seed suite. We obtain these six new versions by applying the following transformers: `clang -O1`, `clang -O2`, `clang -O3`, `ollvm -fla`, `ollvm -bcf` and , `ollvm -sub`. Thus, the classifier t_i is trained with $36 \times i$ positive samples and this same number of negative samples.

Challenging Mirai Identifiers. Figure 15 shows how the accuracy of two different families of classifiers vary with the growth of the training set. We compare a classifier based on the `cnn` model adapted from Zhang et al. [45] and the random forest model (`rf`). Each challenge consists of 12 positive samples (malware) and 12 negative samples (from SPEC CPU2017). We tried seven different challenges. Each challenge takes the 24 samples, and transforms them using—exclusively—one of the seven transformers used in training. The maximum accuracy is 7.0, in which case the classifier correctly identifies the $7 \times (12 + 12) = 168$ challenges. Figure 15 shows that the accuracy of the classifiers increases with the growth of the training set. Both the classifiers considered in Figure 15 achieve an almost perfect hit rate when trained with all the $7 \times 2 \times 36 = 504$ samples. The `rf` approach, in particular, misses only one out of 168 challenges (a benign file obfuscated with `clang -O1`).

Comparison with Anti-Virus Systems. We have compared the accuracy of the classifiers evaluated in Figure 15 with the anti-viruses available at VirusTotal [39]. VirusTotal aggregates several industry-quality anti-virus systems into a single scanner, which can be tried online. We do not know how many different anti-viruses are used by VirusTotal—its website claims that it scans binaries with over 70 implementations [39]. Figure 16 summarizes results for the 168 challenges used to construct Figure 15. We have tried VirusTotal in two ways: first, asking if a software is malware; second, asking if it is Mirai. We show results for the best antivirus, and compare them with our best classifier: the `rf` model trained with 504 samples. The accuracy of our classifier is higher. However, whereas VirusTotal has been engineered to deal with any binary program, our classifier can only tell if a program is a version of Mirai or not.

5 Related Work

Algorithm classification is a form of *code diffing*: the problem of detecting coding patterns within programs. The literature on code diffing is extensive. For an overview, we recommend Section 2 of Ren et al. [34]'s recent work. There are, essentially, two approaches to perform code diffing: semantic analysis and syntactic analysis. The former relies on approaches

[2] Retrieved on August 1st, 2022, from https://github.com/vxunderground/MalwareSourceCode/tree/main/Linux/Mirai-Family

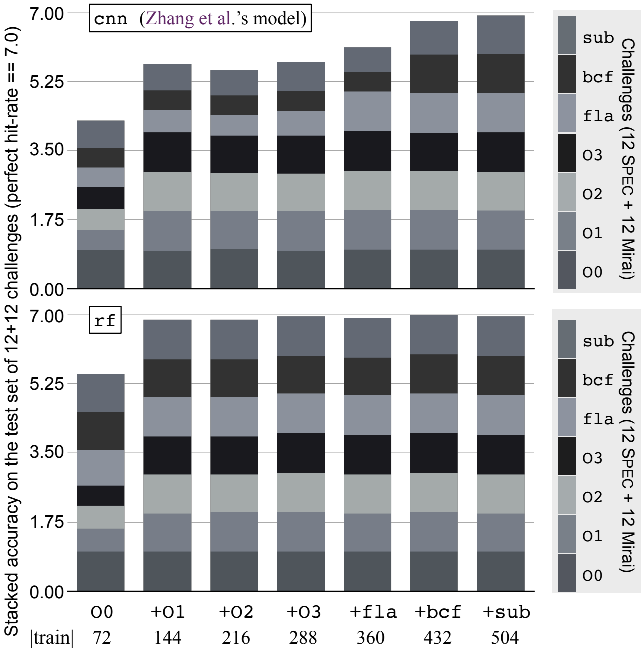


Figure 15. Evaluation of different algorithm classifiers on 168 challenges consisting of different versions of malicious (Mirai) and benign (SPEC CPU2017) codes.

	O0	O1	O2	O3	fla	bcf	sub
is malw.	96.77%	87.09%	83.87%	87.09%	96.77%	90.32%	96.77%
is mirai	77.41%	74.19%	70.96%	74.19%	80.64%	80.64%	80.64%
rf(504)	100.0%	95.83%	100.0%	100.0%	100.0%	100.0%	100.0%

Figure 16. Comparison between the best classifier in Figure 15 and the best antivirus in VirusTotal.

such as symbolic execution/theorem proving [25, 26] or *postmortem* analysis of execution traces [7, 41] to determine similarities between programs. The latter—syntactic-based approaches—do not assume any semantic knowledge about the target programming language. Thus, they tend to be computationally cheaper. There are also hybrid code diffing techniques, which combine semantic and syntactic methodologies. Algorithm classification is typically solved via exclusively syntactic-based approaches [16, 18, 24, 43].

The Four Games in Perspective. As pointed out by Ren et al. [34], most of the code diffing literature focus on the construction of classifiers; hence, they would be described as instances of **Game0**. New program embeddings evaluated as **Game0** include Inst2Vec [4], Code2Vec [2], Ir2Vec [40], Asm2Vec [16], and ProGraML [10]. Brauckmann et al. [5] and Siow et al. [37] use **Game0** to compare the effectiveness of abstract syntax trees, sequences of tokens, and a graph-based embedding. In previous work [11], we have also used **Game0** to compare 13 variations of program embeddings. However, to make our experiments practical, we have used only five classes of algorithmic problems.

Work that aims at building evaders tends to focus on **Game1**, where the classifier has no previous knowledge of the transformations allowed to the evader. Such is the case of Ren et al. [34]’s BinTuner, for instance. David *et al.* [13, 14] perform experiments that we classify as variations of **Game3**. They try to perform code diffing in binaries produced by different compilers. To ease this task, David et al. use optimizations to put binaries into a canonical form. In this sense, the evader would be a different compiler, not some code-obfuscation approach.

6 Conclusion

This paper has compared state-of-the-art program classifiers and evaders on a framework formed by four adversarial games. The current implementation of this framework lets us evaluate 9 × 6 × 2 classifiers, whose design varies according to the representation used to encode programs (nine embeddings); the stochastic model used to classify programs (six implementations) and the code normalization approach pre-applied onto programs (two optimizers). This framework has allowed us to evaluate nine evaders, which differ in how they obfuscate programs to hide their purposes. In hindsight, some of our observations were expected, such as the fact that optimizers are equally as effective as obfuscators to evade classification [34]; or the fact that compiler optimizations can revert naïve code obfuscation [13]. Yet, we found some of our empirical findings surprising. For instance, in symmetric games, we have observed that histograms of opcodes are as effective as program embeddings recently designed to do algorithm classification. We also could not observe advantages of recent models proposed for stochastic algorithm classifications over off-the-shelf machine learning libraries. Finally, knowledge of the obfuscator improves substantially the ability of a classifier to categorize programs (**Game2**), but discovering the obfuscator applied onto a program seems to be a hard task (Sec. 4.7). All this said, our study has limitations. In particular, most of our conclusions have been drawn from experiments performed on a single dataset, formed by 104 programming problems, each with 500 samples. The only different dataset that we have evaluated consists of 48 actual malware plus 48 benign programs of similar sizes. Perhaps, observations derived from different datasets could lead to conclusions other than those enumerated in this paper.

Data Availability Statement. Software artifact [12] is available at https://doi.org/10.5281/zenodo.7374649.

Acknowledgement

The authors would like to thank CNPq, FAPEMIG and CAPES for financial support. The authors are also indebted to the CGO reviewers and Michael O’Boyle, this paper’s shepherd, for many useful suggestions and comments.

A Artifact Appendix

A.1 Abstract

This artifact compares different program classification techniques, and pit them against different evasion techniques. In total, this artifact let us evaluate nine program encoding techniques; seven code obfuscation passes; and seven stochastic classification models. The artifact consists of a docker container with accompanying scripts to replicate Figures 5-15 automatically, plus the dataset and accompanying instructions to replicate Figure 16 manually. For a more up-to-date version of this source code, check https://github.com/lac-dcc/yali.

A.2 Artifact Check-List (Meta-Information)

- **Program:** Seven code classification models: two from Zhang et al. [45]: "dgcnn" (Deep Graph Convolution Neural Network), "cnn" (Convolutional Neural Network); and five from `SciKit` [31]: "rf" (Random Forest); "svm" (Support Vector Machine); "knn" (K-Nearest Neighbors); "lr" (Logistic Regression); and "mlp" (Multilayer Perceptron). One code obfuscator: `O-LLVM` [20]. Optimization passes in clang.
- **Compilation:** clang, gcc, cmake.
- **Dataset:** Training data samples are included.
- **Run-time environment:** Any operating system that supports Docker, Docker-compose, Python3, Wget, Tar, and Sed.
- **Hardware:** Any x86-64 machine with at least 64 GB of RAM memory.
- **Metrics:** Accuracy, F1-Score, Memory (GB), and time.
- **Output:** Jupyter notebooks replicating Figures 5-15.
- **How much disk space required (approx.)?:** 80 GB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** All the experiments take approximately 19 days:
  - Speedup analysis (Figure 13) takes $\sim \frac{23*Rounds}{60}$ hours. We ran 10 rounds, so it takes ~ 4 hours
  - Memory analysis (Figure 7) takes $\sim \frac{14*Rounds}{60}$ hours. We ran 10 rounds, so it takes ~ 2 hours and 20 minutes
  - Game 0 (Figure 7) takes $\sim \frac{14*Rounds}{60}$ hours. We ran 10 rounds, so it takes ~ 2 hours and 20 minutes
  - Game 1 and Game 2 (Figure 8 and Figure 9) take $\sim \frac{112*Rnds}{60}$ hours. We ran 10 rounds, so each one of them takes ~ 18 hours and 30 minutes
  - Game 3 (Figure 11) takes $\sim \frac{98*Rounds}{60}$ hours. We ran 10 rounds, so each one of them takes ~ 16 hours and 20 minutes
  - The experiment to reproduce Figure 12 takes $\sim \frac{10*Rounds}{60}$ hours. We ran 10 rounds, so it takes ~ 1 hour and 40 minutes
  - The experiment to reproduce Figure 15 takes $\sim \frac{10*Rounds}{60}$ hours. We ran 10 rounds, so it takes ~ 1 hour and 40 minutes
  - The experiment to reproduce Figure 14 takes $\sim Rounds * 4$ minutes. We ran 10 rounds, so it takes ~ 40 minutes
  - Comparison between program representations (Figure 5) takes $\sim Rounds * 1.7$ days. We ran 10 rounds, so it takes ~ 17 days
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPL-3.0.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.7374649

A.3 Description

A.3.1 Delivered.

https://doi.org/10.5281/zenodo.7374649

A.3.2 Hardware Dependencies.

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

A.3.3 Software Dependencies.

Docker, Docker-compose, Python3, Jupyter Notebook, Wget, Tar, and Sed. The Docker image contains all the other dependencies to reproduce it.

A.3.4 Data Sets.

- "OJClone": POJ-104 dataset used by Mou et al. [27].
- "BCF": The OJClone dataset obfuscated by OLLVM's Bogus Control Flow.
- "FLA": The OJClone dataset obfuscated by OLLVM's Control Flow Flattening.
- "SUB": The OJClone dataset obfuscated by OLLVM's Instructions Substitution.
- "OLLVM": The OJClone dataset obfuscated by all the OLLVM's passes.
- "MCMC": The OJClone dataset obfuscated by the Monte Carlo approach of Zhang et al. [46].
- "DRLSG": The OJClone dataset obfuscated by the Deep Reinforcement Learning Sequence Generation of Zhang et al. [46].
- "RS": The OJClone dataset obfuscated by the Random-Search strategy of Zhang et al. [46].
- "dataset1", "dataset2", "dataset3", "dataset4": They are a set of programs from OJClone originally produced with clang -O0 and then transformed via either optimizations or obfuscations. There are ten classes of code transformers in each of them.
- "Mirai": 48 different versions of the Mirai malware [19], plus 48 benign programs of similar sizes.

A.4 Installation

1. Download and unpack the zip file from https://doi.org/10.5281/zenodo.7374649. You will get a folder called yali-main.
2. Copy the `.env.example` file[3] into a new file called `.env`.
3. Run the script `setup.sh`, e.g., (`$> ./setup.sh`) to prepare the environment.

A.5 Experiment Workflow

To execute the experiments, run the script `run.sh` as follows:

```
$ ./run.sh all
```

[3] This is a hidden file located in the project's root directory.

It is possible to execute the experiments separately. Below we show how to generate data for specific figures:

- Figure 5: `$> ./run.sh embeddings`
- Figure 7 (second chart) and 12: `$> ./run.sh resource`
- Figure 7 (first chart): `$> ./run.sh game0`
- Figure 8: `$> ./run.sh game1`
- Figure 9: `$> ./run.sh game2`
- Figure 10: This figure is generated by a Jupyter Notebook
- Figure 11: `$> ./run.sh game3`
- Figure 13: `$> ./run.sh speedup`
- Figure 14: `$> ./run.sh discover`
- Figure 15: `$> ./run.sh malware`
- Figure 16: The data in this figure cannot be reproduced automatically, more information in section A.6.1.

A.6 Evaluation and Expected Results

The Statistics folder contains Jupyter Notebooks that plot the data generated by the experiments. Each notebook describes the chart(s) it contains and provides the steps to generate this data. The artifact contains the following notebooks:

- **EmbeddingResults:** Presents comparison of program embeddings in **Game0** using 32 problems. Notebook replicates Figure 5.
- **GameResults:** Presents information about the 4 games proposed in our work and the Figure 15. Notebook replicates Figures 7 (first chart), 8, 9, 11, Figure 15 and 12.
- **ResourceResults:** Presents information about resource consumption (memory and time) of each model. Notebook replicate Figure 7 (second chart).
- **StrategiesResults:** Notebook replicates Figures 10, 14 and 13, with analyses of each strategy.

A.6.1 Reproducing Figure 16. The only figure in the paper that cannot be reproduced automatically is Figure 16, because it requires us submitting files to a website that classifies them as either malicious or benign software. To reproduce Figure 16, we recommend the following steps:

1. Go to www.virustotal.com
2. Click "Choose File" and select file to upload from https://github.com/lac-dcc/yali/tree/main/MalwareDataset/mirai (malicious code) or https://github.com/lac-dcc/yali/tree/main/MalwareDataset/spec_cpu_2006_range (benign code).
3. Retrieve detection score from the upper left bullet. This is the malware detection rate.
4. Retrieve detection labels in front of the names of each anti-virus in the central window.
5. Count the ratio of anti-virus (AVs) assigning the Mirai label to the malicious samples. This is the family classification rate.
6. Repeat for every file.

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 (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[3] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *ICSM ('98)*. IEEE Computer Society, USA, 368.

[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, NY, USA, 3589–3601.

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

[6] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. 2010. Language-Independent Clone Detection Applied to Plagiarism Detection. In *SCAM*. IEEE Computer Society, USA, 77–86. https://doi.org/10.1109/SCAM.2010.19

[7] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. 2012. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *CCS*. Association for Computing Machinery, New York, NY, USA, 169–182. https://doi.org/10.1145/2382196.2382217

[8] 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.

[9] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing Through Deep Learning. In *ISSTA*. ACM, New York, NY, USA, 95–105. https://doi.org/10.1145/3213846.3213848

[10] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2021. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. *CoRR* abs/2109.08267 (2021), 12 pages. arXiv:2109.08267

[11] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintao Pereira, Nilton Luiz Queiroz Junior, and Otavio Oliveira Napoli. 2022. Program representations for predictive compilation: State of affairs in the early 20's. *Journal of Computer Languages* 73, null (dec 2022), 101171–101185. https://doi.org/10.1016/j.cola.2022.101171

[12] Thais Damasio, Michael Canesche, Vinicius Pacheco, Anderson Faustino, Marcus Botacin, and Fernando Pereira. 2022. *A Game-Based Framework to Compare Program Classifiers and Evaders - Artifact*. Universidade Federal de Minas Gerais. https://doi.org/10.5281/zenodo.7374649

[13] 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

[14] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS*. Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3173162.3177157

[15] Breanna Devore-McDonald and Emery D. Berger. 2020. Mossad: Defeating Software Plagiarism Detection. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 138 (nov 2020), 28 pages. https://doi.org/10.1145/3428206

[16] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *SP*. IEEE, Washington, DC, USA, 472–489. https://doi.org/10.1109/SP.2019.00003

[17] Anderson Faustino. 2021. Graphs Based on IR as Representation of Code: Types and Insights. In *SBLP*. ACM, New York, USA, 75–82. https://doi.org/10.1145/3475061.3475063

[18] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *ASE*. Association for Computing Machinery, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480

[19] Harm Griffioen and Christian Doerr. 2020. *Examining Mirai's Battle over the Internet of Things*. Association for Computing Machinery, New York, NY, USA, 743–756. https://doi.org/10.1145/3372297.3417277

[20] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *International Workshop on Software Protection*. IEEE, Washington, DC, US, 3–9.

[21] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.* 28, 7 (jul 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[22] Saruhan Karademir, Thomas Dean, and Sylvain Leblanc. 2013. Using Clone Detection to Find Malware in Acrobat Files. In *CASCON* (Ontario, Canada). IBM Corp., USA, 70–80.

[23] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, Washington, DC, US, 75–86.

[24] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. 2018. ADiff: Cross-Version Binary Code Similarity Detection with DNN. In *ASE*. Association for Computing Machinery, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[25] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *FSE*. Association for Computing Machinery, New York, NY, USA, 389–400. https://doi.org/10.1145/2635868.2635900

[26] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-Based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *SEC*. USENIX Association, USA, 253–270.

[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] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

[29] Lawton Nichols, Kyle Dewey, Mehmet Emre, Sitao Chen, and Ben Hardekopf. 2019. Syntax-Based Improvements to Plagiarism Detectors and Their Evaluations. In *ITiCSE* (Aberdeen, Scotland Uk) *(ITiCSE '19)*. Association for Computing Machinery, New York, NY, USA, 555–561. https://doi.org/10.1145/3304221.3319789

[30] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. 2019. Structural and Nominal Cross-Language Clone Detection. In *FASE (Lecture Notes in Computer Science, Vol. 11424)*, Reiner Hähnle and Wil M. P. van der Aalst (Eds.). Springer, Germany, 247–263. https://doi.org/10.1007/978-3-030-16722-6_14

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. 2011. Scikit-Learn: Machine Learning in Python. *J. Mach. Learn. Res.* 12, null (nov 2011), 2825–2830.

[32] 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

[33] 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, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *NeurIPS Datasets and Benchmarks*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). Curran Associates Inc., Red Hook, NY, USA, 21 pages.

[34] 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

[35] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.2307/1990888

[36] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* 49, 1, Article 4 (apr 2016), 37 pages. https://doi.org/10.1145/2886012

[37] Jing Kai Siow, Shangqing Liu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2022. Learning Program Semantics with Code Representations: An Empirical Study. https://doi.org/10.48550/ARXIV.2203.11790

[38] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. *A Human Study of Comprehension and Code Summarization*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3387904.3389258

[39] https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors. 2022. VirusTotal. VT Intelligence: Combine Google and Facebook and apply it to the field of Malware. https://www.virustotal.com/gui/intelligence-overview

[40] 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

[41] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *ASE*. IEEE Press, USA, 319–330.

[42] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F. P. O'Boyle. 2022. Bind the Gap: Compiling Real Software to Hardware FFT Accelerators. In *PLDI*. Association for Computing Machinery, New York, NY, USA, 687–702. https://doi.org/10.1145/3519939.3523439

[43] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-Based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS*. Association for Computing Machinery, New York, NY, USA, 363–376. https://doi.org/10.1145/3133956.3134018

[44] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *BWCCA*. IEEE Computer Society, USA, 297–300. https://doi.org/10.1109/BWCCA.2010.85

[45] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the AAAI Conference on Artificial Intelligence* (New Orleans, Riverside, USA) *(AAAI '32)*. Association for the Advancement of Artificial Intelligence, New York, NY, USA, 4423–4445.

[46] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2021. Challenging Machine Learning-based Clone Detectors via Semantic-preserving Code Transformations. arXiv:2111.10793 https://arxiv.org/abs/2111.10793

Received 2022-09-02; accepted 2022-11-07