On the Precision of Dynamic Program Fingerprints Based on Performance Counters

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

Marcelo Borges Nogueira
UFRN
Natal, Brazil
marcelo.nogueira@ufrn.br

Sérgio Queiroz de Medeiros
UFRN
Natal, Brazil
sergio.medeiros@ufrn.br

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

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

***Abstract*—Task classification is the challenge of determining whether two binary programs perform the same task. This problem is essential in scenarios such as malware identification, plagiarism detection, and redundancy elimination. Classification can be performed statically or dynamically. In the former case, the classifier analyzes the binary image of the program, whereas in the latter it observes the program's execution. Recent research has demonstrated that dynamic classification is more accurate, particularly in adversarial settings where programs may be obfuscated. This remains true even when both classifiers use the exact representation of programs, such as histograms of instruction opcodes. The superior accuracy of dynamic classification stems from its ability to disregard dead code inserted during the obfuscation process. However, state-of-the-art dynamic techniques, such as Valgrind plugins, can slow down program execution by as much as 100 times due to binary instrumentation. This paper proposes to eliminate this overhead by replacing program instrumentation with the sampling of hardware performance counters. Our findings reveal both advantages and limitations of this approach. On the positive side, classifiers based on hardware counters impose almost no runtime overhead while retaining greater accuracy than purely static classifiers, particularly in the presence of obfuscation. On the downside, counter-based classifiers are slightly less accurate than instrumentation-based approaches and offer coarser granularity, being limited to whole-program classification rather than individual functions. Despite these limitations, our results challenge the conventional belief that dynamic code classifiers are too costly to be deployed in environments such as online servers, operating systems, and virtual machines.**

***Index Terms*—Security, Binary Diffing, Code Classification**

I. Introduction

Task classification is the problem of determining whether two programs implement solutions to the same task. This problem is fundamental to challenges such as plagiarism detection, malware identification, and code lifting. In general, task classification is undecidable, as it amounts to proving program equivalence [8, 37]. As a result, most approaches to task classification are stochastic in nature. In these stochastic approaches, a classifier translates each program into a vector of numbers, using these vectors to compare different programs. Such vectors, henceforth referred to as *embeddings*, can be generated in various ways. For instance, they may encode program features, such as the number of back edges and basic blocks [30], or they may count the frequency of low-level instructions [2] or high-level AST nodes [31].

Recently, [11] demonstrated that effective classifiers can be built using embeddings based on the frequency of instructions executed during a typical run of the program. For example, if a program causes the `add` opcode to be fetched five times and the `jump` opcode to be fetched three times during execution, its embedding would be a vector with five and three in the positions corresponding to `add` and `jump`, respectively. We refer to embeddings constructed from such runtime behavior as *dynamic program fingerprints*, in contrast to embeddings derived from static representations, such as LLVM modules or binary object files. Dynamic fingerprints tend to be more resistant against obfuscation techniques that insert dead code into programs, as such code is never executed [11]. However, generating these fingerprints is expensive. For instance, CFGGrind [38], a Valgrind plugin used to build instruction histograms, can slow down programs by 100 times [11]. In this paper, we investigate whether it is possible to eliminate this overhead without compromising the accuracy of the dynamic program classifier.

a) Embeddings based on Performance Counters: A hardware performance counter is a special register in a processor that counts specific low-level events, such as instructions executed, cache misses, or branch mispredictions, while a program is running. Most modern processor architectures provide hardware performance counters, including x86/x86-64 (Intel and AMD), ARM, POWER, and RISC-V. Access to performance counters is commonly provided through software tools and APIs. On Linux systems, for instance, the `perf` tool is used for this purpose. For portable access, the Performance API (PAPI) [32] abstracts counter usage across platforms. Vendor-specific tools also exist, such as Intel `VTune`, AMD `uProf`, and ARM's `perfmon` interface.

In this paper, we show that hardware performance counters, also known as performance monitoring units (PMUs), can be

979-8-3315-9288-2/26 © 2026 IEEE

used to construct fingerprints that are effective for program classification. These embeddings are vectors whose dimensions represent the values of different performance counters, as explained in Section II. Intuitively, classification based on PMU vectors is expected to be less precise than classification based on histograms of executed instruction opcodes (e.g., consider "this execution fetched five `add` and three `jump` instructions" vs "this execution ran in eight CPU cycles"). On the other hand, PMU-based classification is expected to be faster, as there is virtually no overhead in probing performance counters. Our goal is to quantify this potential loss in precision and to measure the runtime overhead introduced by this alternative classification method.

b) Summary of Findings: We have implemented a program classifier based on hardware performance counters (PMUs). As Section III explains, this classifier embeds programs into a vector space where each dimension corresponds to a different performance counter. In this space, closer programs are more likely to solve the same task than distant programs. Section IV evaluates the PMU-based fingerprint on two task classification problems: algorithmic classification and algorithmic matching. In the first, the goal is to identify the task a given program is designed to solve from a set of possible tasks. In the second, the objective is to match programs from two sets such that each matched pair solves the same task. Experiments on five processors from two architectures (Intel and AMD) led to the following observations:

- **Precision:** As Sections IV-A and IV-G show, the PMU-based classifier achieves slightly higher accuracy (5–20%) than static classifiers that use histograms of assembly or LLVM instructions. Its accuracy is slightly lower (5–10%) than that of classifiers based on dynamic execution traces.
- **Performance:** The runtime of programs with and without performance monitoring is very similar. As Section IV-B demonstrates, for all practical purposes, generating embeddings via PMUs introduces negligible overhead. This stands in contrast to classifiers based on dynamic instruction traces, which can incur slowdowns of up to 100x.
- **Obfuscation:** In adversarial settings, where an evader may obfuscate programs before classification or matching, the PMU-based classifier retains the robustness of dynamic approaches. Furthermore, it substantially outperforms purely static classifiers in terms of precision under obfuscation, as explained in Sections IV-C and IV-D.
- **Selection:** A selection of 15 standard counters focusing on memory access patterns, branch behavior, and instruction retirement achieves accuracy comparable to larger counter sets. Evidence discussed in Sections IV-E and IV-F suggests that this set can be further reduced, depending on the classification task and the target architecture.

II. Overview

This section introduces the program classification challenges that we address (Sec. II-A) and the different types of code embeddings that we study (Sec. II-B).

A. Program Classification Games

Program classification can be formulated in various ways. In this paper, we focus on two complementary problems: *algorithmic classification* and *algorithmic matching*. We define these problems formally below.

Definition 2.1 (Algorithmic Classification): Let $P = \{P_1, P_2, \ldots, P_n\}$ be a collection of problem definitions . Each problem $P_i \in P$ is defined by a set of input/output pairs. Given a program p, the algorithmic classification problem asks: which $P_j \in P$ does p solve?

Algorithmic classification is a standard benchmark in prior work on code understanding and similarity detection [12, 10, 21, 31]. It plays an important role in tasks such as automatic grading, plagiarism detection, and malware categorization, where the goal is to determine the intended functionality of a program. In academic work, this problem is often evaluated using program datasets derived from Programming Online Judges (OJ). This preference for OJ benchmarks stems from a few factors. First, the datasets are naturally organized into problems, each containing numerous solution submissions. Second, these solutions exhibit significant variation as they are implemented by different individuals. Third, each problem is accompanied by a textual description, test inputs, and expected outputs, which are valuable for code understanding tasks.

Definition 2.2 (Algorithmic Matching): Let $Q = \{q_1, q_2, \ldots, q_n\}$ and $R = \{r_1, r_2, \ldots, r_n\}$ be two sets of programs, where each $q_i \in Q$ solves a distinct problem, and for every $q \in Q$, there exists a corresponding $r \in R$ solving the same problem. The algorithmic matching problem asks for a list of pairs (q, r) with $q \in Q$ and $r \in R$, such that q and r solve the same task.

Algorithmic matching is commonly used to evaluate *binary diffing* techniques. This task is vital in scenarios such as vulnerability detection, software provenance, and patch analysis, where analysts seek to identify semantically equivalent binaries compiled under different settings. Typical benchmarks include program sets compiled from the same source using different compilers, optimization levels, or obfuscation schemes [11, 15, 23, 25, 27, 49, 51, 39]. Nevertheless, algorithmic classification can also be evaluated under an *adversarial* setting [12, 21], where a *classifier* is challenged by an *evader*. In an adversarial context, the evader applies transformations to programs to obscure their semantics while preserving functionality.

B. Code Embeddings

Current approaches to program classification work by converting programs into numerical vectors through *embedding functions*, as formalized in Definition 2.3. These embeddings are constructed or learned representations that capture structural and semantic aspects of programs. They can be derived from various types of program representations, such as source code, control-flow graphs, abstract syntax trees, or binary instructions.

Definition 2.3 (Embedding Function): An *embedding function* maps a program representation (e.g., a sequence of tokens, AST nodes, LLVM instructions, binary instructions, etc) to a vector in $\mathbb{R}^n$. The resulting vector is referred to as the *program's embedding*.

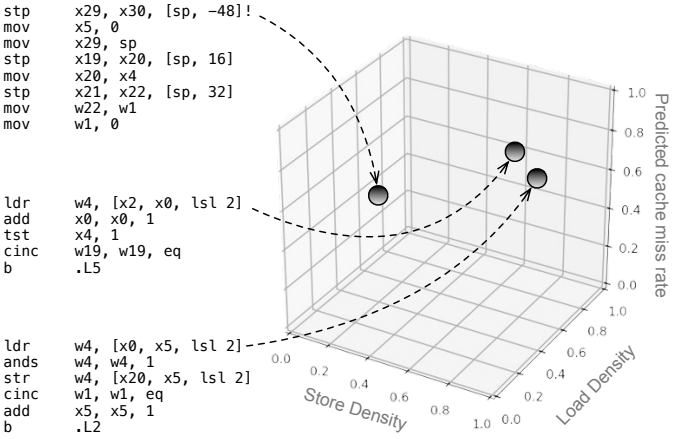

Fig. 1. A simple two-dimensional embedding of basic blocks based on store and load densities. Each point represents a basic block, and its coordinates reflect the proportion of store and load instructions relative to the block's total instruction count. In this example, the embedding is being used as a predictor of cache behavior.

(a)

```c
#include <stdio.h>

int main() {
    long long sum = 0;
    for (int i = 0; i<100000000; i++) {
        sum += i;
    }
    printf("Sum = %lld\n", sum);
    return 0;
}
```

(b)

```
$> gcc -O2 -o sum sum.c
$> perf stat -e cycles, \
          instructions, \
      cache-references, \
   branch-instructions ./sum
```

(c) Performance counter stats for './sum':

```
     1,034,527      cycles
       794,438      instructions          # 0.77  insn per cycle
        14,582      cache-references
       161,217      branch-instructions

       0.001482336 seconds time elapsed
```

Fig. 2. Example use of `perf stat` to collect hardware performance counters for a simple C program running on Linux Ubuntu (kernel 5.15.0-139) on an Intel(R) Xeon(R) CPU E5-2680 v2 at 2.80GHz.

Embeddings are used in program classification by transforming programs into fixed-length numerical vectors that capture their structural or behavioral characteristics. In algorithmic classification, a classifier, such as a neural network or nearest-neighbor model, uses these embeddings to predict which task a given program is designed to solve, based on similarity to embeddings of known programs. In algorithmic matching, embeddings enable comparison between two sets of programs by measuring distances in the embedding space; programs that solve the same task are expected to have similar embeddings and thus lie close to each other. This vector-based representation enables classification and matching to be framed as geometric problems, allowing for the use of efficient statistical and machine learning techniques.

Example 2.1: Figure 1 shows a simple embedding function. In this hypothetical scenario, the embedding function maps basic blocks to vectors in a two-dimensional space. The first dimension of this space represents the *store density*, defined as the number of store instructions divided by the total number of instructions in the basic block. The second dimension represents the *load density*, calculated similarly for load instructions. This embedding captures coarse memory access patterns of each basic block. For instance, a block with a high load density and low store density might correspond to a read-heavy loop, while a block with high store density might indicate intensive write operations. Such representations could be used as features in a model to predict the likelihood of cache misses in different regions of a program. Of course, this prediction would have limited accuracy, as cache behavior also depends on additional factors such as memory access stride, cache associativity, and interference from other code.

There are many ways to construct embedding functions. Some are based on simple, hand-engineered features, such as the store/load density vectors illustrated in Example 2.1. Others use richer program characteristics, such as histograms of token types, AST node frequencies, or instruction opcodes, which capture the syntactic or structural patterns of the code. More sophisticated embeddings are learned using machine learning models. For example, neural networks can be trained to map programs to vector spaces using techniques like autoencoders [45], graph neural networks applied to control-flow or data-flow graphs [9, 5], or models inspired by natural language processing, such as BERT-style encoders trained on large corpora of code [41]. In the next section, we introduce a simple embedding based on hardware performance counters.

III. PMU-Based Classification

This section describes an implementation of an embedding function based on performance-monitoring units. It introduces hardware performance counters (Sec. III-A); describes the construction of an embedding function that maps program execution to vectors (Sec. III-B); explains how these embeddings can be used to classify programs (Sec. III-C); and finally discusses a few practical considerations when using counters, such as how to build embeddings per function (Sec. III-D).

A. PMU-Based Classification: An Intuition

Performance counters can be accessed through various APIs and tools, as discussed in Section I. On Linux systems, the most common interface is the `perf` subsystem, which offers both command-line utilities and low-level system calls. Example 3.1 demonstrates the usage of `perf`.

Example 3.1: Figure 2 illustrates how hardware performance counters can be used to profile a simple C program that computes the sum of the first 100 million integers in a loop. In this example, the Linux `perf stat` tool collects four hardware events: the number of CPU cycles, the number of executed instructions, the number of cache references, and the number of branch instructions. The output reveals, for instance, that the program executed nearly 800,000 instructions, of which approximately 160,000 were branches. Metrics such as the instructions-per-cycle ratio (IPC), also reported by `perf`, help characterize the processor's efficiency given a workload.

Even when two programs, P_0 and P_1, execute a similar number of instructions, their dynamic behavior may differ due to differences in the algorithms they implement. Aspects such

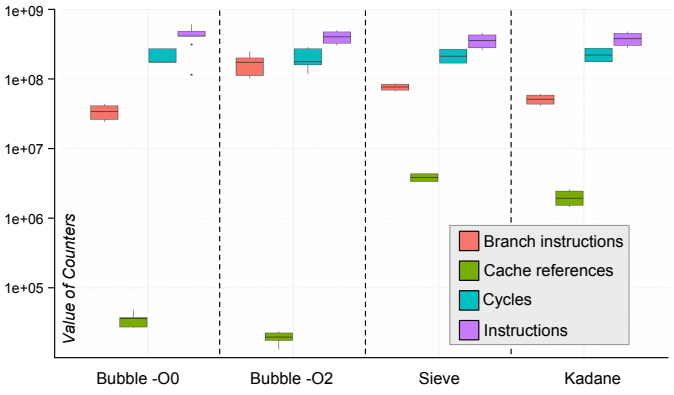

Fig. 3. Hardware counter values collected for ten implementations of three algorithms, running on Linux Ubuntu (5.15.0-139) on an Intel(R) Xeon(R) E5-2680 v2 @ 2.80GHz.

as memory access patterns, control-flow complexity, and cache utilization are shaped by a program's logic and structure. For example, one program may cause frequent memory accesses, resulting in a higher number of cache references or page faults. At the same time, another may involve unpredictable control flow, leading to more branch misses. Hardware performance counters capture these low-level events, offering an approximate view of runtime behavior. Thus, a vector of performance counters can act as a "*fingerprint*" of a program's execution, making it possible to distinguish between algorithms, even if their instruction counts are similar. This idea is illustrated in Example 3.2.

Example 3.2: Figure 3 presents values for the four counters from Example 3.1 across four different executables running on an Intel E5 processor. These executables implement three distinct algorithms: Bubble Sort, the Sieve of Eratosthenes, and Kadane's Algorithm for the Maximum Subarray Problem [4]. Each algorithm was implemented by ten different programmers, with samples collected from Moodle, an online learning management system. The binaries were executed to run approximately the same number of instructions. Although their execution times are similar in terms of CPU cycles, the observed behaviors differ significantly in terms of branch instructions and cache references. Such variations are also visible within implementations of the same algorithm compiled with different compilers. For instance, the figure includes fingerprints of Bubble Sort compiled with `gcc -O0` and `gcc -O2`.

B. The PMU-Based Embedding Function

As we shall explain in Section V, most previous work on binary diffing uses exclusively static embeddings. In contrast, dynamic embeddings are constructed from features observed during the program's execution, and therefore require both the program and a representative input [29, 35]. An example of a dynamic embedding is the histogram of executed instruction opcodes [11]. Embeddings based on hardware performance counters also fall into the dynamic category, as they are collected by observing a program's runtime behavior. In this case, the embedding is a numerical vector, where each dimension corresponds to the value of a specific performance counter measured after running the program on a given input.

Example 3.3: Going back to Example 3.1, Figure 2 (c) shows a possible output produced by `perf` when used to monitor the program in Figure 1 (a). In this example, the embedding is the vector $\langle$`cycles` $= 1845,123,456$, `instructions` $= 1,002,345,678$, `cache_references` $= 12,345,678$, `cache_misses` $= 1,234,567\rangle$

Rationale. Although PMU counters record low-level hardware events (e.g., cache accesses, branches, misses), these events directly reflect the control-flow and data-access behavior of algorithms. Thus, PMUs capture microarchitectural manifestations of program semantics. Algorithms such as bubble sort and Kadane, seen in Example 3.2, may execute a similar number of instructions, yet differ in memory-access locality and branching patterns. These differences are captured by PMUs, leading to distinct fingerprints. Moreover, PMUs exhibit resilience against transformation, as they aggregate the effects of executed code, rather than its syntax. Hence, transformations that modify the layout or add dead code (as in obfuscation) might alter static structure and still leave intact dynamic events.

On the Choice between Absolute vs Relative Values. The PMU-based embedding proposed in this paper relies on the absolute values of hardware counters. This contrasts with earlier work [6], which used relative counter values computed against a baseline version of the same program. In their approach, each dimension of the embedding was defined as the value of a particular counter in the optimized version of a program divided by the corresponding value in the non-optimized version. This representation was natural, given their goal: predicting the impact of compiler optimizations. In our case, however, the objective is different: we aim to distinguish between arbitrary executable versions, including differentiating optimized from non-optimized variants of the same program. This broader goal motivates our use of absolute counter values.

C. On the Implementation of the Games

Figure 4 shows the setup of the different games that Section IV will evaluate. Playing these games requires *Prediction Models*: methods that map program embeddings to predicted categories or labels. For algorithmic classification (see Definition 2.1), we use random forests. For algorithmic matching (see Definition 2.2), we use the *Euclidean distance* between embeddings to compare programs.

1) Algorithmic Classification via Random Forests: To solve the algorithmic classification problem using random forests, performance counters are used to construct the fixed-length numerical vectors mentioned in Section III-B. In our setup (see Fig. 4), the training set consists of 32 problem classes, each represented by 80 programs whose embeddings are labeled with the class (the specific problem $P_j \in P$, as in Definition 2.1) they solve. A random forest classifier is trained on these labeled embeddings. At test time, the classifier is evaluated on a disjoint set of programs: the same 32 classes, but with 20 previously unseen programs per class. Each test program is run with a sample input, its performance counters are collected, and its

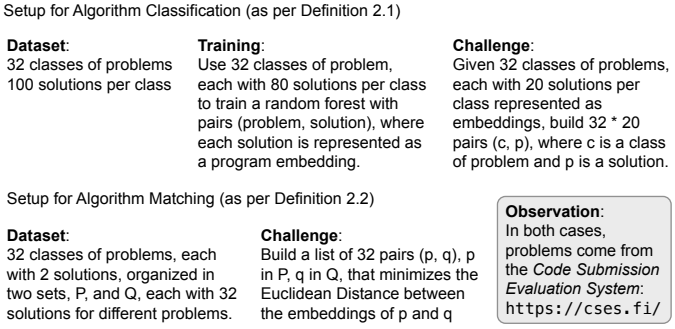

Fig. 4. The experimental setup adopted to evaluate the different games proposed in this paper.

embedding is passed through the trained forest. Each decision tree casts a vote, and the forest predicts the class corresponding to the majority vote. Notice that the same methodology can be applied to any other form of embedding vector. Section IV relies on this observation to compare different embeddings.

Our decision to use random forest was based on findings from previous research; however, variations are possible. For example, prior work has explored algorithmic classification using neural networks [30] and k-nearest neighbors [42]. However, recent evaluations suggest that performance differences among these models are minor, with random forests offering a good balance between accuracy and computational efficiency [10, 12, 21].

2) Algorithmic Matching via Euclidean Distance: To solve algorithmic matching, we group programs by spatial proximity, as Figure 4 explains. Following Definition 2.2, we construct two sets of embeddings: Q and R, each containing 32 programs, with one sample drawn randomly, without repetition, from each of 32 problem classes. Every program in Q has a corresponding program in R that solves the same problem, albeit in a different manner. To identify matching pairs, we compute the Euclidean distance between every pair of embeddings (q_i, r_j), with $q_i \in Q$ and $r_j \in R$. For each q_i, we select the r_j that lies closest in the embedding space; that is, the one with the smallest Euclidean distance, and report this as a predicted match. As in the case of algorithmic classification, the same methodology can be applied to different embeddings. Section IV shall present a comparison between these embeddings.

Also, as seen in the case of algorithmic classification, binary matching could be performed using other distance metrics between vectors, such as *cosine similarity*, *Manhattan distance*, *Mahalanobis distance*, or *Jaccard distance*, depending on properties of the embedding. While [14] recommends cosine similarity when evaluating the ASM2VEC embedding, [11] reports a slight advantage of the Euclidean distance over cosine similarity in the context of binary matching.

D. Practical Considerations

When setting up the experiments presented in Section IV, we adopted two simplifications, which are discussed in this section. First, as we explain in Section III-D1, we run each program only once, when reading hardware performance counters. Second, as seen in Section III-D2, we classify whole programs, not parts of them, such as functions or individual modules.

1) On the Sampling Methodology: Hardware performance counters are not deterministic: multiple executions of the same program might yield different results. In an offline setting, PMU-based embeddings can be produced from multiple executions of the same program. In an online setting, where classification is performed on-the-fly, only one measurement is possible. This is the setup that Section IV evaluates. Thus, we emphasize that the results reported in this paper are based on a single observation of the program behavior, and note that multiple executions are likely to yield more accurate results. Nevertheless, we avoid repeating each execution various times, as that would increase the project's cost in terms of energy consumption, processing time, and researcher time.

In this sense, prior work has shown that performance counter variations across multiple runs of the same program are generally small [13, 43, 47, 46, 50]. We confirm these findings in our own experiments, which show minimal variability across repeated runs. In Section IV, we use a relatively large number of programs: 100 per problem class. Thus, residual noise is averaged out across the dataset. Finally, it is worth noting that even multiple executions cannot fully eliminate noise caused by factors such as background system activity, cache warm-up effects, or scheduling artifacts; in some cases, they may even introduce new sources of variability.

2) On the Granularity of the Embedding Function: This paper utilizes performance counters collected throughout the entire execution of a program to construct embeddings for program-level classification or matching. We opted for this methodology because it is easy to implement and evaluate. Nevertheless, it is possible to reduce the granularity of the approach, for instance, by classifying individual functions instead of entire programs [3], but with caveats.

To collect performance counters at the function level, one can instrument the program to reset and read hardware counters at function entry and exit points. Tools like Intel's BOLT (a post-link binary optimizer) are capable of generating function-level profiles by using performance sampling data. BOLT, for example, collects performance events such as instruction counts and branch statistics, then aggregates them per function to guide code layout optimizations. A similar strategy can be used to construct embeddings that describe the dynamic behavior of individual functions. This approach requires precise control over counters during execution, for instance, via low-level APIs like `perf_event_open`.

However, this function-level approach presents limitations. First, short-lived functions may not accumulate enough events to produce meaningful or stable embeddings. Second, isolating counters per activation may introduce overhead or noise, particularly if functions are called frequently or recursively. Finally, architectural and OS-level restrictions, such as limited or absent PMU access on some platforms (e.g., Apple Silicon), can make fine-grained profiling infeasible or even impossible.

IV. Evaluation

This section evaluates the following research questions:

- RQ1: What is the relative accuracy of the different classifiers when used on a non-adversarial setting?
- RQ2: What are the static and dynamic overheads of the different classifiers?
- RQ3: What is the impact of code obfuscation on the accuracy of the different classifiers?
- RQ4: What is the accuracy of classification in an adversarial setting if classifiers are trained on obfuscated code?
- RQ5: How does the accuracy of the zero-overhead dynamic classifier change on different machines?
- RQ6: How does the selection of hardware performance metrics influence the accuracy of program classification, and how does this relationship change in adversarial and non-adversarial scenarios?
- RQ7: How do the different embeddings compare on binary matching?

a) Hardware: To demonstrate that PMU-based classifiers can be effectively implemented in different processors, this section uses two collections of hardware. Accuracy results are reported on the following processors:

- A1: Ryzen Threadripper 3960X 24-Core at 2.2 GHz, with 64 GB of RAM, running Linux Ubuntu 20.04.
- A2: Ryzen 9 7900X3D 12-Core at 3 GHz, with 64 GB of RAM, running Linux Ubuntu 20.04.
- I1: Xeon E5-2630 at 2.60 GHz, with 128 GB of RAM, running Linux Ubuntu 20.04.

The performance results reported in Section IV-B use two machines that have been prepared to run performance experiments with minimal external interference:

- I2: Core i5-7500 4-Core at 3.40GHz with 16 GB of RAM, running Linux Ubuntu 22.04.5.
- I3: Core i5-2400 4-Core at 3.10GHz with 8 GB of RAM, running Linux Ubuntu 22.04.5.

Additionally, machine I2 is also used to experiment with the variability of the performance counters, a study that Section IV-A reports.

b) Performance Counters: The Intel and AMD processors do not feature the same collection of performance counters. However, there exists an intersection of 15 counters that are available in both platforms. These counters track typical performance metrics such as cache accesses, branch predictions, instruction retirements, TLB misses, and cycle counts. Henceforth, we shall use only these 15 counters when building program embeddings.

c) Code Classifiers: In addition to the PMU-based classifier that this paper proposes, this section considers the following embeddings, which were taken from previous work [11, 19, 44]:

- IR2Vec: the static embedding [44]. IR2Vec encodes LLVM intermediate representation tokens into dense numerical vectors based on data- and control-flow context. This embedding is built in a two-step process. First, a vocabulary (the encoding of different instructions and program properties) is trained; then the program is mapped into such a vocabulary. This embedding, like the counter-based, is not based on some notion of a histogram of instructions.
- `S-LLVM`: (Static LLVM View) histogram of LLVM instructions in the compiler's intermediate representation.
- `S-x86`: (Static Binary View) histogram of x86 instructions in the `.text` section of binaries, produced via the `Capstone` Disassembler v5.0 [18].
- `D-x86`: (Dynamic Binary View) histogram of instructions observed via CFGGRind [38], during an execution of a program. Each execution of the same instruction type is counted.
- `H-x86`: (Hybrid, e.g., Dynamic Program Slice) histogram of instructions observed via CFGGrind [38] that form the control-flow graph of the program parts covered during execution. Thus, multiple executions of the same type of instructions are counted only once.

A. RQ1: Accuracy on a Non-Adversarial Setting

For the non-adversarial classification game of Definition 2.1, we use the evaluation setup discussed in Section III-C1: classification via random forests, and a dataset formed by 32 programs, each with 100 solutions, out of which 80 are separated for training, and 20 for testing.

Discussion: Figure 5 subsumes the results of this section. It shows that dynamic embeddings consistently outperform static approaches. CFGgrind dynamic histograms achieve the highest accuracy, ranging from 91% to 95%, surpassing static IR2Vec by 26–33% and the LLVM histogram by 20–26%. PMU-based histograms follow closely, with accuracies ranging from 83% to 86%, outperforming the x86 histogram by 10–16%. The gap across embeddings is however not extreme: all methods, static or dynamic, achieve hit rates above 50%, while a random classifier would succeed only $1/32 = 3.125\%$ of the time.

Figure 5 also shows that the performance of dynamic embeddings is largely independent of the underlying hardware. This outcome was expected, as all three machines provide the same set of 15 performance counters. In contrast, compiler optimizations do have a noticeable impact. Except for IR2Vec, accuracy tends to improve as the optimization level increases. This observation is consistent with prior work [11, 12]. In particular, [45] has recently demonstrated that compiler optimizations exert a normalizing effect on binary code, thereby enhancing the accuracy of classifiers.

On the Variability of Counter Values. The variability of PMU readings did not meaningfully affect classification accuracy, primarily because the variation itself is very small. To quantify this effect, we executed all 3,200 programs in our dataset four times on machine I2 and computed, for each metric, the pairwise differences across runs (six differences per program). We then examined the median of these differences: the metric *Faults* showed the lowest variability (0.03%), whereas *iTLB-load-misses* exhibited the highest (13.4%). Across all metrics, the median of medians was 2.1%. These findings align with prior observations [43, 50], indicating that PMU variability

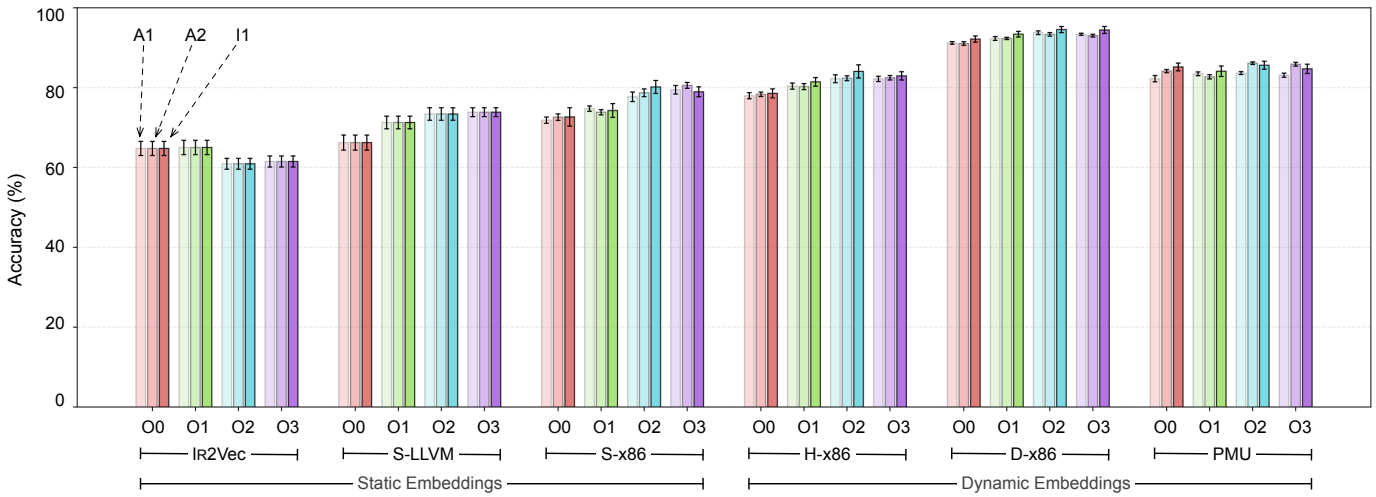

Fig. 5. Accuracy of different classifiers when used in a non-adversarial setting, where binary programs are not obfuscated before being given to the classifier. For reference, a random classifier is expected to have an accuracy of $1/32 = 3.125\%$.

remains low and stable across execution environments. For example, in a non-adversarial scenario in which all programs were compiled with `-O2`, running 16 repeated classifications led to accuracy variations of only 1.4

B. RQ2: Runtime Overhead

The runtime overhead of reading performance counters is extremely low. To measure it, we performed the same experiment in two dedicated Intel machines (see I2 and I3 in the list of hardware). These machines are designed to minimize external factors and ensure accurate measurements; they are not connected to a network and do not run a graphical user interface (GUI) server. Before a measurement, we restart the system and wait for a warm-up phase. We compare the program's runtime with and without hardware counter sampling. The reported overhead associated with hardware counter sampling represents a worst-case scenario, as it corresponds to sampling 23 Intel events, plus tracking four time events using the eight physical registers available on the machine. Experiments in the other sections, which use a smaller set of 15 counters, are expected to exhibit a lower overhead. Notice that we use `perf` in `stat` mode; hence, there is no sampling rate. Rather, `perf stat` operates in counting mode, which means it directly reads the performance counters at the beginning and end of the target program.

Discussion: Figure 6 shows the median of the runtime of programs with and without the use of `perf` to collect performance data. The figure refers to the I2 machine; however, I3 shows very similar behavior. For perspective, the figure also includes the runtime when CFGGRIND is used to collect instruction histograms, as this is a required step of the other dynamic classifiers that we evaluate. The contrast in overhead is stark: while CFGGRIND increases the average program runtime by more than 30×, the overhead introduced by `perf` is almost negligible and mostly constant for each machine. Across all 3,200 programs (32 sets, each with 100 programs),

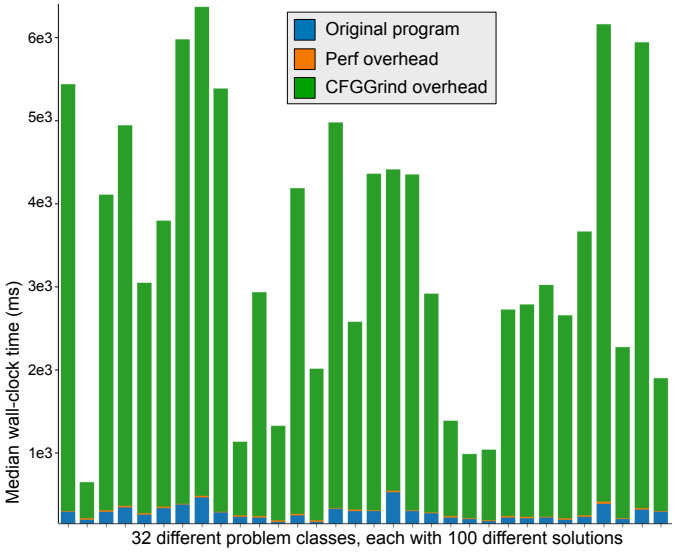

Fig. 6. Runtime overhead of different dynamic program classifiers in machine I2.

the total execution time for I2 and I3 was, respectively, 447 and 526 s without `perf` and 495 and 596 s with it. Although this difference is statistically significant, with a Mann-Withney's U-test [26, 48] yielding a p-value inferior to 0.001, it isn't essential according to the corresponding Cliff's delta [7, 40]. Due to space constraints, we omit data for the other machines used in this paper; however, we confirm that their behavior is also very similar to that observed in Figure 6.

C. RQ3: The Impact of Obfuscation

For the adversarial version of the Classification Game of Definition 2.1, we employ the three semantics-preserving transformations in `O-LLVM` [24]: bogus control flow (BCF), which inserts never-taken branches; control-flow flattening (FLA), which rewrites the control-flow graph into a single

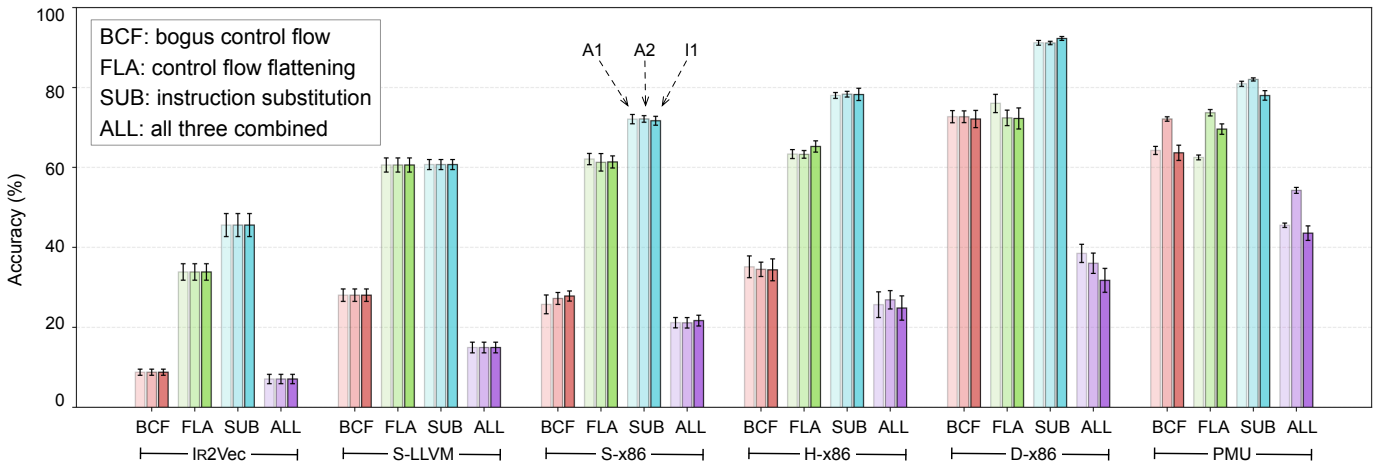

Fig. 7. Relative accuracy of different classifiers in an adversarial setting where programs are obfuscated before classification.

loop within a switch statement; and instruction substitution (SUB), which replaces operations with equivalent instructions. In this setting, classifiers are trained on programs compiled with `clang -O0` and tested on their obfuscated counterparts.

Discussion: Obfuscation reduces accuracy across all embeddings, as shown in Figure 7, with losses ranging from 10% under simpler transformations to nearly 80% under combined attacks. Dynamic embeddings, such as the dynamic histogram, demonstrate stronger resilience, maintaining 70%–90% accuracy under BCF, FLA, and SUB, although they fall to about 35% under ALL: a 55% average reduction across machines. PMU-based classification follows a similar pattern: 60%–80% for isolated obfuscations and 45%–50% for ALL, outperforming static approaches by 20%–40% on average. By contrast, static embeddings degrade sharply: IR2VEC drops below 10% for BCF and ALL; LLVM histograms fall to 15%–30%; and x86 histograms remain around 20%–30%, with overall reductions exceeding 50% under combined techniques.

These findings highlight the superior robustness of dynamic embeddings, which retain 20%–30% more accuracy than static embeddings under adversarial conditions. This aligns with previous observations [11], who noted that execution-centric features are better at preserving semantic signals despite structural code modifications. The most striking conclusion, however, is the relative resilience of PMU-based classification. Against the strongest adversary—combining BCF, FLA, and SUB—the PMU-based classifier achieved the best accuracy among all methods. This result is particularly noteworthy given that, as discussed in Section IV-B, PMU-based profiling imposes virtually no runtime overhead, in stark contrast to Valgrind-based approaches, which can slow execution by up to 100×.

D. RQ4: Symmetric Classification with Obfuscation

Previous work has shown that classifier accuracy can be recovered when training is performed directly on obfuscated programs [12]. This outcome is expected, as it effectively removes the adversary from the setting. Nevertheless, obfuscation still increases code entropy and thereby reduces the overall accuracy of classification engines [45]. In this section, we evaluate the performance of the PMU-based classifier in a high-entropy scenario, where both training and testing programs are obfuscated using the various techniques available in `O-LLVM`.

Discussion: Figure 8 summarizes the results for RQ4. Dynamic embeddings exhibit substantial improvements when training is aligned with obfuscation, achieving accuracies comparable to non-adversarial baselines and significantly surpassing the low results observed when clean data was used for training. For example, the CFGgrind dynamic histogram achieves approximately 88% for BCF and 86% for ALL, representing gains of more than 50% compared to mismatched training. Performance counter histograms also recover strongly, reaching around 80% for ALL, with uplifts of over 30% from prior adversarial lows. Their reliance on execution profiles allows classifiers to capture and generalize obfuscated behaviors, effectively transforming vulnerabilities into strengths under matched datasets.

Static and hybrid embeddings likewise benefit from aligned training, though they remain less effective overall. IR2VEC improves to roughly 41% for ALL, a recovery of more than 30% from its near-random performance in mismatched setups. Similarly, LLVM and x86 histograms rise to approximately 54% and 55%, respectively, gaining around 40%.

Unlike in Section IV-C, the PMU-based classifier does not surpass CFGgrind in this symmetric setting. Still, it remains the second most accurate method, outperforming the different static approaches and the hybrid classifier by a significant margin. Notably, the hybrid classifier also incurs very high runtime overheads (the `H-x86` and `D-x86` classifiers use the same infrastructure to produce embeddings), sometimes slowing programs by factors of over 100×, whereas PMU-based profiling is essentially performance-free.

E. RQ5: The Impact of the Target Architecture

The accuracy of the PMU-based classifier appears to be independent of the target architecture. Overall, we observe that

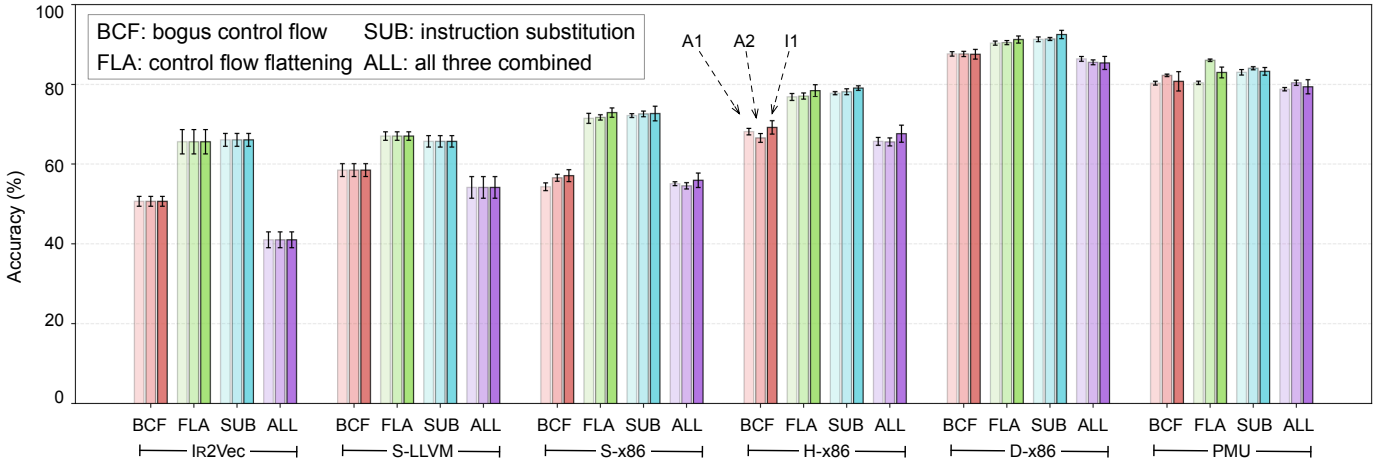

Fig. 8. Accuracy of classifiers trained and tested on obfuscated programs.

this classifier is more accurate than `H-x86` and slightly less precise than `D-x86`, although both exhibit significantly higher overheads. Nevertheless, the choice of hardware may have a minor impact on this accuracy, as discussed in this section.

Discussion: The **Ir2Vec** and **LLVM-based classifiers** are completely architecture-neutral. Consequently, among the static classifiers, only the `x86`-based classifier, which uses embeddings based on a histogram of assembly instructions, is influenced by the target hardware. However, we observed minimal variation in the accuracy of this classifier due to the architecture. Similarly, we also found minimal variation in the accuracy of the CFGgrind-based classifiers, which also analyze histograms of `x86` instructions, albeit produced in a different manner. These observations lead us to conclude that in a non-adversarial setting, the impact of the target architecture on the accuracy of these classifiers is negligible.

Furthermore, in a non-adversarial setting, we did not observe a significant impact on the accuracy of the **PMU-based classifiers**. Depending on the chosen optimization level, any of the three target architectures (A1, A2, and I1) might yield the highest performance. However, in the adversarial setting, we noted higher accuracy on the Ryzen 9 (A2) architecture. These results, as shown in Figure 7, are statistically significant. It is essential to note that this performance difference is absent when using the other classifiers (Figure 5) or when classifiers are also trained on obfuscated programs (Figure 8). The lower variance on the A2 architecture might explain the higher accuracy of the PMU-based classifier in this environment.

F. RQ6: Impact of Metric Selection on Classification Accuracy

In this paper, we use embeddings based on 15 counters: *L1-dcache-load-misses*, *L1-dcache-loads*, *branch-instructions*, *branch-load-misses*, *branch-loads*, *branch-misses*, *cache-misses*, *cache-references*, *cycles*, *dTLB-load-misses*, *dTLB-loads*, *faults*, *iTLB-load-misses*, *iTLB-loads*, and *instructions*. As already explained, we chose these 15 counters because they are common to both the AMD and Intel architectures available in our experiments. An immediate question that this choice raises is whether our results would be different had we used the other counters available on each architecture. This section shows that the answer is negative.

Discussion: Our Intel and AMD machines give us 23 and 25 counters, respectively. The intersection of these sets contains 15 counters. We could not observe statistically significant differences in accuracy, per architecture, when using the extra counters. In other words, using only the 15 standard counters achieved the same accuracy as the full 25 pool in the AMD setting. This observation suggests that it is possible to reduce the number of counters even further; however, in this case, the results depend on the architecture and the game (whether it is adversarial or not). In every setting, the best-performing subset varies in size from 5 to 10 counters, and there is no single "best 10" that dominates across all patterns. Instead, each pattern yields its own optimal sub-selection (e.g., branch metrics for control-heavy binaries vs. TLB and L1 metrics for memory-bound ones). A consistent finding is that L1-related counters are present in all the most accurate embeddings.

Category-Level Analysis. To address the counter-selection challenge, we conducted a study to determine which families of performance counters contribute most reliably to accuracy. We analyzed situations when counters were used in isolation and when compared against the full 15-counter set. Table I summarizes the results for the adversarial setting on machine A1.

TABLE I
ACCURACY CONTRIBUTION (%) BY CATEGORY ON MACHINE A1 IN AN ADVERSARIAL SETTING; EX.: THE VALUE +6.85 INDICATES THAT ADDING THE BRANCH COUNTER INCREASES ACCURACY BY THIS MARGIN WHEN PROGRAMS ARE OBFUSCATED WITH BOGUS CONTROL FLOW.

Category	BCF	FLA	SUB	ALL
Branch	+6.85	+1.30	+5.93	-3.48
Cache	+3.18	+3.64	+1.19	+8.71
Cycles	+0.98	-1.56	+2.17	-3.83
Cycles+Instr	-1.96	-3.12	+0.99	-13.59
Faults	+8.80	+8.05	+2.96	+11.50
Instr	-1.47	+1.30	-1.78	-8.01
TLB	+5.13	+1.56	+0.20	+0.70
Accuracy	0.64	0.60	0.79	0.45

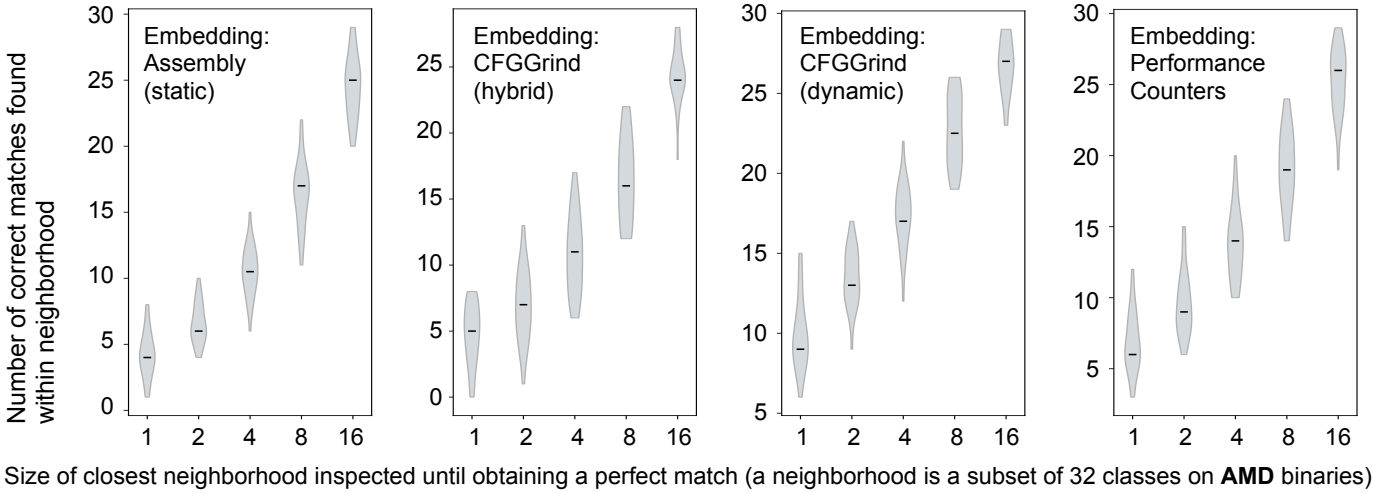

Fig. 9. Results of algorithmic matching on AMD binaries. This experiment is conducted in a non-adversarial setting where programs in Q and R are compiled with `clang -O1`. The x-axis represents the size of the top-K neighborhood considered for matching, while the y-axis indicates the number of correct matches out of 32 possibilities.

Memory-centric counters—*Faults*, *Cache*, and *TLB*—yield the most consistent and architecture-agnostic improvements across obfuscation schemes. In contrast, cycle- and instruction-related metrics frequently reduce accuracy, suggesting that they inject noise or redundancy rather than offering discriminative value. This outcome is expected: the raw number of executed instructions is often unrelated to the underlying algorithmic structure. Branch-related counters show mixed effects, helping with highly control-flow–oriented algorithms but harming accuracy in other patterns, indicating that they are informative only in specific scenarios.

These findings refine the counter-selection problem. Instead of exploring the full combinatorial space, practitioners can begin with memory-focused categories as a dependable baseline, selectively add Branch counters for control-intensive workloads, and systematically exclude cycle and instruction metrics. This approach dramatically narrows the search space and accelerates the identification of effective 5–10 counter subsets tailored to each architecture and obfuscation scenario. In practice, an adaptive methodology emerges: retain the 15-counter portable core for cross-platform deployment, derive scenario-specific subsets for each optimization/obfuscation pattern, and combine them using stable ratios or lightweight learning techniques. Such adaptive counter selection approaches the accuracy of CFGGrind-based dynamic embeddings while maintaining the near-zero overhead of PMU sampling.

G. RQ7: Binary Matching

The previous experiments focused on algorithmic classification (Definition 2.1). This section evaluates the performance of different embeddings in the context of algorithmic matching (Definition 2.2). Our methodology proceeds as follows:

1) We randomly draw 32 pairs of programs, one per problem class, to form two sets, Q and R, each containing 32 programs.
2) For each $q \in Q$, we identify its match $r \in R$ as the program whose embedding yields the smallest Euclidean distance among all 32×32 pairwise comparisons.
3) We repeat steps (1) and (2) 100 times, to minimize statistical noise.

We also evaluate a relaxed version of this methodology using a *top-K neighborhood* criterion: a match is considered correct if the true pair lies among the K closest embeddings. For example, with $K = 8$, the match is successful if the correct program is one of the eight nearest neighbors in R.

Discussion: Figure 9 shows the results of this experiment in a non-adversarial setting, where both sets Q and R were compiled using `clang -O1`. In all cases, we observe a consistent ordering among the embeddings. Dynamic histograms collected with CFGGrind achieve the highest number of perfect matches, followed closely by PMU-based embeddings. PMU-based matching consistently outperforms both CFGGrind hybrid histograms and static embeddings based on x86 assembly instruction frequencies. Notably, at a neighborhood size of 16, PMU-based embeddings correctly match 26 out of 32 programs, while CFGGrind dynamic embeddings reach 27.

Although additional results are omitted due to space constraints, we confirm that this trend persists across the other three architectures (Intel and AMD) available to us. Furthermore, changes to the compilation pipeline, such as varying optimization levels or applying obfuscation, do not alter the relative performance of the embeddings. Importantly, these results reflect a non-adversarial setup, where Q and R are compiled with identical flags. Under adversarial conditions, such as different compiler settings or injected obfuscations, the accuracy of all embeddings drops significantly, often approaching random matching levels. We have also evaluated algorithmic matching using different embeddings, including LLVM-based histograms and IR2VEC. These embeddings are consistently worse then either the PMU-based embedding or

the embeddings based on CFGGrind dynamic histograms.

V. Related Work

To the best of our knowledge, the use of program embeddings derived from hardware performance counters to solve algorithmic classification or matching is novel. Nonetheless, embeddings based on performance counters have been previously explored to select compiler optimizations. Furthermore, algorithmic classification and matching are longstanding problems in computer science, with a rich history of research.

A. Algorithm Classification and Matching

The problems of algorithmic classification (Definition 2.1) and algorithmic matching (Definition 2.2) have been widely studied, both for their practical applications, such as malware detection and plagiarism identification, and as benchmarks for program understanding. The ability to classify programs according to the algorithms they implement is often taken as evidence of a system's capability to understand code. A notable contribution in this space is by Mou *et al.* [31], who formulated algorithmic classification as a neural network problem. However, the origins of the problem predate this work: the theoretical foundations can be traced back to the work of Rice [37], while Copeland *et al.* have explored its practical implementations. Most solutions to algorithmic classification are static, as noted in recent surveys [1, 10, 22, 28]. Static approaches have the advantage of being independent of program inputs and do not require executing potentially malicious code. Dynamic approaches, on the other hand, tend to be more resilient against adversarial obfuscation techniques. Aligned with this intuition, the findings in this paper indicate that since PMU counters reflect effective execution, not static code layout, they allow for achieving greater accuracy over static embeddings under obfuscation.

B. Embeddings Based on Performance Counters

One of the main inspirations for this work is the work of Cavazos *et al.* [6], who demonstrated how embeddings derived from performance counters can be used to select optimal compiler optimizations for a program. Their technique was designed for speed, enabling its use in just-in-time compilation. This idea sparked a line of research in which performance counters have been used for tasks such as predicting program performance on specific architectural configurations [16, 17] and selecting compiler optimizations [33, 34]. Like Cavazos *et al.*, we also value fast profiling. However, our focus differs: prior work has primarily targeted optimization selection, whereas we apply performance-counter-based embeddings to the problem of program understanding, with the specific goal of algorithmic classification.

C. Energy-Based Dynamic Classification

Recently, Queiroz *et al.* [36] proposed using power dissipation as a means to classify programs based on the problems they solve. They employ power (the ratio of energy consumption to execution time) as a program fingerprint. Energy measurements are obtained via the RAPL (Running Average Power Limit) registers, which are available in modern Intel and recent AMD architectures. Since reading RAPL registers incurs negligible overhead, this approach aligns with our goal of distinguishing programs dynamically without incurring significant performance penalties. However, Queiroz *et al.*'s method exhibits limited accuracy: our replication of their work on an Intel i5-7500 yields only 62.2% (non-adversarial) and 18.6% (adversarial) accuracy. Similar results (47.5% and 16.1%) were observed on an Intel i5-2400, where fewer energy-related metrics were available. In contrast, using the performance counters from Section IV, we achieve 88.3%/43.6%, and 89.1%/43.9% on exactly the same setup.

VI. Conclusion

This paper showed that hardware performance counters enable very low-overhead dynamic fingerprints for accurate algorithm classification and matching. Across architectures, PMU embeddings outperform static methods by 10–40%, whereas instrumentation-based dynamic embeddings slow execution by 30–100 times. Our key findings reveal that PMUs maintain robust performance even under adversarial conditions, retaining 20-30% greater accuracy than static embeddings when programs are obfuscated. While PMU-based classification slightly trails the precision of fine-grained instruction histograms (by 5-10%), this gap is offset by their practical advantages: low instrumentation cost, architectural portability, and scalability to real-world environments where overhead is prohibitive. These findings challenge the conventional wisdom that dynamic program analysis is too costly for production environments.

We believe that the findings of this paper open several promising directions for future work. A natural extension is to explore finer-grained PMU-based embeddings, including function-level or phase-level profiling, potentially augmented with low-overhead mechanisms such as Processor Trace or Last Branch Record data. It is also important to evaluate the robustness of PMU-based fingerprints in concurrent or non-deterministic workloads, where counter variability may be more pronounced. Finally, an open question is whether more advanced learning techniques can further narrow the precision gap with instrumentation-based methods while preserving the low-overhead benefits that make PMU-based classification.

Data Availability Statement

The reproducible artifact for this paper is available at either https://github.com/ComputerSystemsLaboratory/Rouxinol or via Zenodo [20].

Acknowledgment

This work was supported by the AI competence center ScaDS.AI Dresden/Leipzig (01IS18026A-D), by the German Research Council (DFG) through the TransT project (552689849), by FAPEMIG (APQ-00440-23), by CNPq (#444127/2024-0), and by CAPES (PrInt).

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Comput. Surv.*, 51(4), jul 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[3] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, May 1997.

[4] Jon Bentley. Programming pearls: algorithm design techniques. *Commun. ACM*, 27(9):865–873, September 1984.

[5] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *CC*, page 201–211, New York, USA, 2020. ACM.

[6] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, page 185–197, USA, 2007. IEEE Computer Society.

[7] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

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

[9] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O'Boyle, and Hugh Leather. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In *ICML*, volume 139, pages 2244–2253, Baltimore, Maryland, USA, 18–24 Jul 2021. PMLR.

[10] Anderson Faustino da Silva, Edson Borin, Fernando Magno Quintão Pereira, Nilton Luiz Queiroz Junior, and Otávio Oliveira Napoli. Program representations for predictive compilation: State of affairs in the early 20's. *J. Comput. Lang.*, 73:101171, 2022.

[11] Anderson Faustino da Silva, Jeronimo Castrillon, and Fernando Magno Quintão Pereira. A comparative study on the accuracy and the speed of static and dynamic program classifiers. In *International Conference on Compiler Construction*, pages 13–24, New York, NY, USA, 2025. Association for Computing Machinery.

[12] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. 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 2023, page 108–121, New York, NY, USA, 2023. Association for Computing Machinery.

[13] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Security and Privacy*, pages 20–38. IEEE, 2019.

[14] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *S&P*, pages 472–489, USA, 2019. IEEE Computer Society.

[15] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. Deepbindiff: Learning program-wide code representations for binary diffing. In *NDSS*. The Internet Society, 2020.

[16] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, Grigori Fursin, and Michael F. P. O'Boyle. Portable compiler optimisation across embedded programs and microarchitectures using machine learning. In *MICRO*, page 78–88, New York, NY, USA, 2009. Association for Computing Machinery.

[17] Christophe Dubach, Timothy M. Jones, and Michael F. P. O'Boyle. Exploring and predicting the effects of microarchitectural parameters and compiler optimizations on performance and energy. *ACM Trans. Embed. Comput. Syst.*, 11S(1), June 2012.

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

[19] Anderson Faustino. The rouxinol project – repository and docker container. https://github.com/ComputerSystemsLaboratory/Rouxinol. [Online; accessed 22-Feb-2024].

[20] Anderson Faustino da Silva. On the precision of dynamic program fingerprints based on performance counters. https://doi.org/10.5281/zenodo.17574396, November 2025. [Online; accessed 10-Nov-2025].

[21] Artyom V. Gorchakov, Liliya A. Demidova, and Peter N. Sovietov. Analysis of program representations based on abstract syntax trees and higher-order markov chains for source code classification task. *Future Internet*, 15(9), 2023.

[22] Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3), apr 2021.

[23] Ang Jia, Ming Fan, Xi Xu, Wuxia Jin, Haijun Wang, and Ting Liu. Cross-inlining binary function similarity detection. In *ICSE*, New York, NY, USA, 2024. Association for Computing Machinery.

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

[25] Yao Li, Dawei Yuan, Tao Zhang, Haipeng Cai, David Lo, Cuiyun Gao, Xiapu Luo, and He Jiang. Meta-learning for multi-family android malware classification. *ACM Trans. Softw. Eng. Methodol.*, 33(7), August 2024.

[26] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*,

pages 50–60, 1947.

[27] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

[28] Hou Min and Zhang Li Ping. Survey on software clone detection research. In *ICMSS*, page 9–16, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. Binsim: trace-based semantic binary diffing via system call sliced segment equivalence checking. In *Security Symposium*, SEC'17, page 253–270, USA, 2017. USENIX Association.

[30] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for binary optimization. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[31] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *AAAI*, page 1287–1293, Palo Alto, CA, US, 2016. AAAI Press.

[32] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. In *Proceedings of the HPCMP users group conference*, volume 710, Washington, DC, USA, 1999. Citeseer, Department of Defense.

[33] Girish Mururu, Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. Generating robust parallel programs via model driven prediction of compiler optimizations for non-determinism. In *ICPP*, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Eunjung Park. *Automatic selection of compiler optimizations using program characterization and machine learning*. PhD thesis, University of Delaware, 2015.

[35] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, page 709–724, USA, 2015. IEEE Computer Society.

[36] Sergio Queiroz de Medeiros, Marcelo Borges Nogueira, and Gustavo Quezado. Investigating the energy consumption of c++ and java solutions mined from a programming contest site. *Journal of Computer Languages*, 84:101341, 2025.

[37] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[38] Andrei Rimsa, José Nelson Amaral, and Fernando M. Q. Pereira. Practical dynamic reconstruction of control flow graphs. *Softw. Pract. Exp.*, 51(2):353–384, 2021.

[39] Andrei Rimsa, Anderson Faustino da Silva, Camilo Santana, and Fernando Magno Quintão Pereira. Binary diffing via library signatures. In *International Symposium on Code Generation and Optimization (CGO)*, pages 1–11. IEEE, 2026.

[40] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *Annual meeting of the Florida Association of Institutional Research*, volume 177, page 34, 2006.

[41] Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo. An exploratory study on code attention in bert. In *ICPC*, page 437–448, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Clóvis Daniel Souza Silva, Leonardo Ferreira da Costa, Leonardo Sampaio Rocha, and Gerardo Valdísio Rodrigues Viana. KNN applied to PDG for source code similarity classification. In *BRACIS*, pages 471–482. Springer, 2020.

[43] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, pages 59–67. IEEE, 2008.

[44] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. IR2VEC: LLVM IR based scalable program embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), 2020.

[45] S. VenkataKeerthy, Soumya Banerjee, Sayan Dey, Yashas Andaluri, Raghul PS, Subrahmanyam Kalyanasundaram, Fernando Magno Quintão Pereira, and Ramakrishna Upadrasta. Vexir2vec: An architecture-neutral embedding framework for binary similarity, 2024.

[46] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *Symposium on Workload Characterization*, pages 141–150. IEEE, 2008.

[47] Vincent M Weaver, Dan Terpstra, and Shirley Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *ISPASS*, pages 215–224. IEEE, 2013.

[48] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[49] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi. Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection. In *DSN*, pages 224–236, Los Alamitos, CA, USA, jun 2021. IEEE Computer Society.

[50] Dmitrijs Zaparanuks, Milan Jovic, and Matthias Hauswirth. Accuracy of performance counter measurements. In *Performance analysis of systems and software*, pages 23–32. IEEE, 2009.

[51] Anshunkang Zhou, Yikun Hu, Xiangzhe Xu, and Charles Zhang. ARCTURUS: Full coverage binary similarity analysis with reachability-guided emulation. *ACM Trans. Softw. Eng. Methodol.*, 33(4), apr 2024.