PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model

Xiangzhe Xu*
Zhou Xuan*
xu1415@purdue.edu
xuan1@purdue.edu
Purdue University
West Lafayette, USA

Shiwei Feng
Purdue University
West Lafayette, USA
feng292@purdue.edu

Siyuan Cheng
Purdue University
West Lafayette, USA
cheng535@purdue.edu

Yapeng Ye
Purdue University
West Lafayette, USA
ye203@purdue.edu

Qingkai Shi
Purdue University
West Lafayette, USA
shi553@purdue.edu

Guanhong Tao
Purdue University
West Lafayette, USA
taog@purdue.edu

Le Yu
Purdue University
West Lafayette, USA
yu759@purdue.edu

Zhuo Zhang
Purdue University
West Lafayette, USA
zhan3299@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

ABSTRACT

Binary similarity analysis determines if two binary executables are from the same source program. Existing techniques leverage static and dynamic program features and may utilize advanced Deep Learning techniques. Although they have demonstrated great potential, the community believes that a more effective representation of program semantics can further improve similarity analysis. In this paper, we propose a new method to represent binary program semantics. It is based on a novel probabilistic execution engine that can effectively sample the input space and the program path space of subject binaries. More importantly, it ensures that the collected samples are comparable across binaries, addressing the substantial variations of input specifications. Our evaluation on 9 real-world projects with 35k functions, and comparison with 6 state-of-the-art techniques show that PEM can achieve a precision of 96% with common settings, outperforming the baselines by 10-20%.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**.

KEYWORDS

Binary Similarity Analysis, Program Analysis

ACM Reference Format:
Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616301

*Both authors contributed equally to this research.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616301

1 INTRODUCTION

Binary similarity analysis determines if two given binary executables originate from the same source program. It has a wide range of applications such as automatic software patching [3, 29, 34, 39, 40, 49], software plagiarism detection [6, 27, 38, 42, 52], and malware detection [4, 7, 14, 15, 18, 21, 51]. For example, assume a critical security vulnerability has been reported and fixed in a library. It is of prominent importance to apply the patch to other deployed projects that included the library. However, the library may be compiled with different settings in different projects. Binary similarity analysis allows identifying all the variants. Given a pool of candidate binaries, which are usually functions in executable forms, a similarity analysis tool reports all the binaries in the pool equivalent to a *queried binary*. The problem is challenging as aggressive code transformations such as loop unrolling and function inlining in compiler optimizations may substantially change a program and produce largely different executables [36].

Given its importance, there is a large body of existing work. Earlier work (e.g., [5, 19]) focuses on extracting static code features such as control-flow graphs and function call graphs. They are highly effective in detecting binaries that have small variations. Many proposed to use dynamic information instead [9, 12, 16, 43] because it better discloses program semantics. For example, *in-memory-fuzzing* (IMF) [43] uses fuzzing to generate many inputs and collects runtime information when executing the program on these inputs. It then uses the collected information to compute binary similarities. When the fuzzer can achieve good coverage, IMF is able to deliver high-quality results. However, achieving good coverage is difficult for complex programs (see our example in Section 2.1). Recently, Machine Learning and Deep Learning techniques are used to address the binary similarity problem [23, 28, 30, 31, 44, 50, 52]. These techniques work by training models on a large pool of binaries that have positive and negative samples. The former includes binaries compiled from the same source and the latter includes those that are functionally different. The models are hence supposed to learn (implicit) features that can be used to cluster functionally equivalent programs. However, as shown in Sections 2.2 and 4.2, these models may learn features that are not robust, and in many cases, not semantics oriented, leading to sub-optimal results.

Inspired by the existing works that leverage dynamic information [12, 16, 43, 54], we consider the semantics of a binary to be a distribution of its inputs and their corresponding *externally observable values* during executions. Observable values are those encountered in I/O operations, and global/heap memory accesses. Compared to other runtime values such as those in registers, observable values are persistent across automatic code transformations as compilers hardly optimize these behavior [16, 43]. However, since we need to compare arbitrary binaries, ideally, we would have to collect sufficient samples in the input space of all these binaries. Making such samples universally comparable is highly challenging. In Section 2.2, we show that a naive sampling strategy that executes all subject binaries on the same set of seed inputs can hardly work as different binaries take inputs of different formats. For example, a valid input for a program A is very likely an invalid input for programs B and C. As such, it can only trigger similar error handling logics in B and C, making them not distinguishable.

In this paper, we propose a sampling technique that can effectively approximate semantics distributions by selecting and interpreting a small set of equivalent paths across different versions of a program. It is powered by a novel probabilistic execution engine. It runs candidate binaries on a fixed set of random seed values. Although many of these seed values lead to input errors, it systematically unfolds the program behavior starting from the execution paths of these seed values, called the *seed paths*. Specifically, it flips a bounded number of predicates along the seed paths. For instance, flipping a failing input check forces the binary to execute its normal functionality. While predicate flipping is not new [32, 51, 53], our technique features a *probabilistic sampling algorithm*. Specifically, we cannot afford exhaustively exploring the entire neighborhood (of the seed paths) even with a small bound (of flipped predicates). Hence, we leverage a key observation that the predicates with the largest and the smallest *dynamic selectivity* tend to be stable before and after automatic transformations, while other predicates vary a lot (by the transformations). Dynamic selectivity is a metric computed for a predicate instance that measures the distance to the decision boundary. For example, assume a predicate `x>y` yields true, `x-y` denotes its dynamic selectivity. Our theoretical analysis in Section 3.5 discloses that since automatic transformations cannot invent new predicates, but rather remove, duplicate, and reposition them, the likelihood that code transformations change the ranking of predicates with the smallest/largest selectivity is much smaller than that for other predicates. Hence, we sample paths by flipping predicates that have close to the largest and the smallest selectivity, following the *Beta-distribution* [17] that has a U shape, biasing towards the two ends. Therefore, if two binaries are equivalent, our algorithm can sample a set of corresponding paths in the binaries by flipping their corresponding predicates such that the observable values along these paths disclose the equivalence.

Our contributions are summarized as follows.

- We propose a novel probabilistic execution model that can effectively sample the semantics distribution of a binary and make the distributions from all binaries comparable.
- We develop a path sampling algorithm that is resilient to code transformation and capable of sampling equivalent paths when two binaries are equivalent. We also conduct a theoretical analysis to disclose its essence.
- We propose a probabilistic memory model that can tolerate invalid memory accesses due to predicate flipping while respecting the critical property of having equivalent behavior when the binary programs are equivalent.
- We develop a prototype PEM. We conduct experiments on two commonly used datasets including 35k functions from 30 binary projects and compare PEM with five baselines [12, 28, 30, 31, 43]. The results show that PEM can achieve more than 90% precision on average whereas the baselines can achieve 76%. PEM is also much more robust when the true positives (i.e., binaries equivalent to the queried binary) are mixed with various numbers of true negatives (i.e., binaries different from the queried binary) in the candidate pool, which closely mimics real-world application scenarios. Consequently, PEM can correctly find 7 out of 8 1-day CVEs from binaries in the wild, whereas the baselines can only find 2. We upload PEM at [46].

2 MOTIVATION AND OVERVIEW

2.1 Motivating Example

Our motivating example is adapted from the main function of `cat` in `Coreutils`. The simplified source code is shown in Fig. 1a. Lines 2 to 10 parse the command line options. Lines 12 to 19 iteratively read the file names from the command line and emit the file contents to the output buffer. The function delegates the main operations to two functions. When some conditions at line 13 are satisfied, a simpler method *simple_cat()* is called. Otherwise, it calls a more complex function that formats the output according to the full panoply of command line options. For example, at line 22, if the global flag `print_invisible` is set, the function prints out the ASCII values of invisible characters.

Compiler optimizations may substantially transform a program. In Fig. 2b and Fig. 2a, we show the control flow graphs (CFGs) for our motivating example generated by two respective compilation settings, -O0 meaning no optimization and -O3 meaning having all commonly used optimizations applied. The `switch` statement at line 3 is compiled to hierarchical `if-then-else` structures with -O0, as shown in the orange circle in Fig. 2b. In contrast, it is compiled to an *indirect jump* with -O3, as shown in the orange circle in Fig. 2a. The predicate at line 13 corresponds to the blue circle in Fig. 2b. We can see two branches, each consisting of only one basic block. Two delegated functions are called in the two basic blocks, respectively. However, the two functions are inlined in the optimized version, resulting in branches with much more blocks, e.g., 50 blocks in the branch of the complex function, as shown in the blue circle in Fig. 2a.

```
1  void main_cat(int argc, char** argv){
2      while(-1!=(c=get_cli_opt(argc, argv, "bestuv"))){
3          switch(c){
4              case 'b': flag0 = true; ...; format = true; break;
5              case 'e': flag1 = true; break;
6              case 'v': print("Coreutils v8.30"); break;
7              ...
8              default: quote("error"); abort();
9          }
10     }
11     ... // define: pageSize, inbuf and insize
12     do{
13         if((flag0||flag1) && format){
14             ret = simple_cat(inbuf, insize);
15         }else{ ...
16             outbuf = xmalloc(outbuf, pageSize)
17             ret = complex_cat(inbuf, insize, outbuf);
18         }
19     }while(...);
20 }
21 int complex_cat(char* inbuf, int insize, char* outbuf){
22     if(print_invisible && inbuf[i]<0x20)
23         outbuf[...] = to_ascii(inbuf[i])
24 }
```

(a) Coreutils: cat

```
1  void main_touch(int argc, char** argv){
2      while(-1!=(c=get_cli_opt(argc, argv, "bcdfhv"))){
3          switch(c){
4              case 'b': flag |= 0x100; break;
5              case 'c': flag = 0x1; break;
6              case 'v': print("Coreutils v8.30"); break;
7              ...
8              default: quote("error"); abort();
9          }
10     }
11     for(int i = begin; i < argc; i++)
12         touch_function(argv[i]);
13
14 }
```

(b) Coreutils: touch

Figure 1: Motivating Example

To better illustrate the challenges, we introduce another function adapted from the main function of `touch` in `Coreutils`, as shown in Fig. 1b. The function `touch` modifies the meta information of files. Lines 2 to 10 parse the command line options and the for-loop at line 11 iteratively performs the touch operation. We can see from Fig. 2c that the syntactic structures of `touch` and `cat` are more similar than those between `cat` with and without optimizations. The observation can be quantified by the statistics of these CFGs shown in the caption.

2.2 Limitations of Existing Techniques

Fuzzing-Based Techniques. There are techniques that leverage fuzzing to explore the dynamic behavior of programs and use them in similarity analysis. For example, *in-memory fuzzing* (IMF) [43] iteratively mutates function inputs and collects traces. Since the parameter specifications for functions in stripped binaries are not available, it is challenging to generate inputs that can achieve good coverage. In our example, IMF can hardly generate legal command line options for the function `main_cat`. Thus most collected behavior is from the error processing code at line 8. Moreover, they tend to collect similar (error processing) behavior from `main_touch`. As such, the downstream similarity analysis likely draws the wrong conclusion about their equivalence. Our experiments in Section 4.2 show that IMF can achieve a precision of 76% on complex cases, whereas ours can achieve 96%.

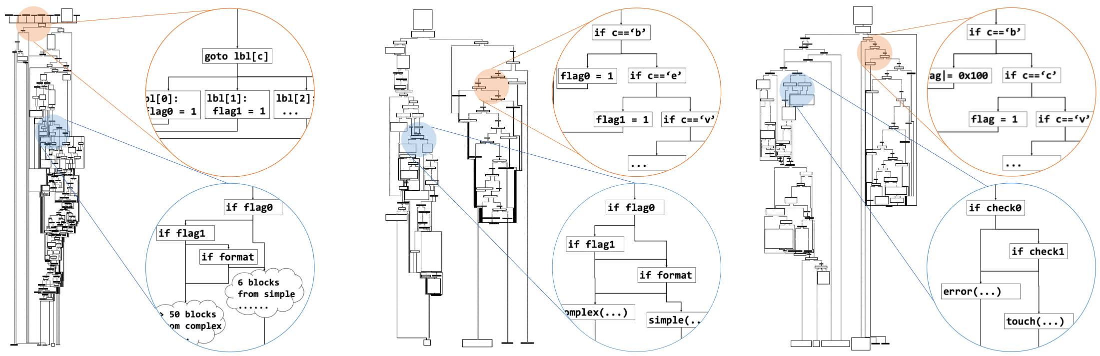
Forced-Execution-Based Techniques. To extract more behavior from binary code, there are methods that use coverage as guidance to execute every instruction in a brute-force fashion. A representative work BLEX [12] executes a function from the entry point. Then it iteratively selects the first unexecuted instruction to start the next round of execution until every instruction is covered. We call techniques of such nature forced-execution-based as they largely ignore path feasibility. There are two essential challenges for these techniques. First, they tend to use code coverage within a function as the guidance for forced execution, which has the inherent difficulty in dealing with function inlining [36]. Another challenge is to provide appropriate execution contexts when execution starts at arbitrary (unexecuted) locations. For example, suppose that in the first few rounds, BLEX executes the true branch at line 14 of Fig. 1. When it tries to cover the false branch at line 17, it uses a fresh execution context, discarding the variables computed at line 11. According to our experiments in Section 4.2, these techniques can achieve a precision of 69%, whereas our technique can achieve 96%.

Learning-Based Techniques. Emerging techniques [30, 31, 45, 52] leverage Machine Learning models. Some models [45, 52] extract static features from CFGs. However, these static features are not robust in the presence of optimizations. Another line of work uses language models [30, 31]. Their hypothesis is that these models could learn instruction semantics and hence function semantics. To limit the vocabulary (i.e., the set of words/tokens supported), binaries are often normalized before they can be fed to models. For example, immediate values (i.e., constants in instructions) and constant call targets are replaced with a special token `HIMM` in SAFE [30], e.g., the token `x_call_HIMM` around line 8 in Fig. 3 (b) and (c) that corresponds to the function invocation `get_cli_opt`. While this makes training convergence feasible, a lot of semantics are lost.

These models may not learn to classify based on instructions essential to function semantics. For example, SAFE leverages an NLP technique called *attention* [41]. Conceptually, the attention mechanism determines which instructions are important to the output. We highlight the statements and their tokens with the largest attention values in Fig. 3. In these three functions, the first few tokens (in gray) with large attention values are in the function prologues. The corresponding instructions (e.g., `push`) perform the same functionality, saving register values to memory and allocating space for local variables. In Fig. 3a, the model also pays attention to tokens/instructions related to the switch-case statement. As discussed before, however, static structures are not reliable due to optimizations. In contrast, in Fig. 3b, the model instead emphasizes the normalized function invocation at line 8, which is not distinguishable from the invocation at line 8 in (c) with a large attention value as well. From the parts that the model pays attention to, it is easy to explain why SAFE concludes `cat@O0` is more similar to `touch@O0`, instead of `cat@O3`. We visualize the weights of full attention layers in Fig. 25 of an extended version of this paper [47].

2.3 Our Technique

We aim to leverage program semantics in similarity analysis. We define the semantics of a binary program P as follows.



(a) Cat@O3 (144 blocks and 218 edges) (b) Cat@O0 (83 blocks and 144 edges) (c) Touch@O0 (89 blocks and 120 edges)

Figure 2: Control-Flow-Graphs for Motivation Examples

```
void main_cat(int argc, char** argv){
    //not meaningful in source code



    while(-1!=(c=get_cli_opt(argc, argv, "bestuv")))
    {
        switch(c){
            case 'b':
                flag0 = true; ...;
                break;
            case 'e':
                flag1 = true;
                break;
            case 'v':
                print("Coreutils v8.30");
                break;
            default: quote("error"); abort();
        }
    }
}
```

```
X_push_r14
X_push_r12
X_sub_rsp,_0x148

X_sub_eax,_0x41
X_cmp_eax,_0x35
X_UNK
X_jmp_rax

X_jmp_0x97
```

(a) cat@O3

```
void main_cat(int argc, char** argv){
    //not meaningful in source code



    while(-1!=(c=get_cli_opt(argc, argv, "bestuv")))
    {
        switch(c){
            case 'b':
                flag0 = true; ...;
                break;
            case 'e':
                flag1 = true;
                break;
            case 'v':
                print("Coreutils v8.30");
                break;
            default: quote("error"); abort();
        }
    }
}
```

```
X_push_rbp
X_sub_rsp,_0x1c8
X_mov_[rbp*1+-12],_edi

X_movabs_rdx,_HIMM
X_call_HIMM
X_cdqe
```

(b) cat@O0

```
void main_touch(int argc, char** argv){
    //not meaningful in source code



    while(-1!=(c=get_cli_opt(argc, argv, "bcdfhv")))
    {
        switch(c){
            case 'b':
                flag |= 0x100; ...;
                break;
            case 'c':
                flag = 0x1;
                break;
            case 'v':
                print("Coreutils v8.30");
                break;
            default: quote("error"); abort();
        }
    }
}
```

```
X_push_rbp
X_sub_rsp,_0x148
X_mov_[rbp*1+-12],_edi

X_UNK
X_call_HIMM
```

(c) touch@O0

Figure 3: Our example (Fig. 1) in SAFE. The statements highlighted in yellow have large attention (and hence are important). The gray boxes to the right (of the yellow statements) denote the corresponding tokens. Special token `HIMM` denotes a constant or a constant control flow target.

Definition 2.1. *The semantics of a binary program P is represented by a distribution $(x, OV(P(x))) \sim \mathcal{D}$, with $x \in \mathcal{X}$ an input to P and $OV(P(x))$ the set of externally observable values when executing P on x. Observable values are those observed in I/O operations, global, and heap memory accesses.*

Intuitively, the joint distribution of inputs and observable values when executing P on the inputs denotes P's semantics. Observable values are hardly altered by code transformations.

A Naive Sampling Method. One may not need to collect a large number of samples to model the aforementioned distribution because if two programs are equivalent, executing them on equivalent inputs produces equivalent observable values. Therefore, a naive method is to provide the same set of inputs to all programs such that those that are equivalent must have identical observable value distributions. However, such a simple method is ineffective because of the following reasons. First, even equivalent programs might have different input specifications (e.g., different numbers of parameters and different orders of parameters), making automatically feeding equivalent inputs to them difficult. Furthermore, different programs have different input domains. When the provided inputs are out-of-range (and hence invalid), the corresponding observable value distributions cannot be used to cluster programs. In our example, the valid domain of c at line 3 of `main_cat` is a set of characters *{b,e,s,t,u,v}* whereas the domain of c at line 3 of `main_touch` is *{b,c,d,f,h,v}*. Without input specifications, which are hard to acquire for binary functions, the naive sampling method may provide a random input, say, $c = 173$. As a result, the executions of both functions fall into the exception handling paths and the observable values are not distinguishable.

Our Method. Instead of solving the input specification problem, which is very hard for binary programs, we propose a technique agnostic to such specifications. Specifically, we propose a novel probabilistic execution model that serves as an effective sampling method to approximate the distribution $\mathcal{D}$ denoting program semantics. Given a program P, we acquire its semantic representation as follows. We execute P on a set $\mathcal{X}$ of pre-determined (random) inputs, which is an invariant for all programs we want to represent. To address the challenge of input specification differences, we assign the same value $x \in \mathcal{X}$ to each input variable (for all programs). That is, we feed the same value to all input parameters, making their order irrelevant. We repeat this for all values in $\mathcal{X}$. As an example, for the programs in Fig. 1, we set `argc` and `**argv` (all elements in the buffer) in both `main_cat` and `main_touch`, as well as `*inbuf`, `insize`, and `*outbuf` in function `complex_cat`, to 173, acquiring three executions. Then we set them to 97, acquiring another three executions, and so on.

These random values may not be valid inputs and hence the corresponding executions may not disclose meaningful semantics. We hence further sample *k-edge-off* behavior.

Definition 2.2. *Given a program P and an input x, let p be the program path taken with input x, we say a path p' is k-edge-off (from p) if k predicates along the execution need to be flipped to other branch outcomes in order to acquire p'.*

For instance, suppose that when executing `main_cat` with $c = 173$, the path p is 2-3-8. If the branch at line 2 is flipped to line 11, assuming that the following execution path is 13-16-17-19, 2-11-13-16-17-19 is 1-edge-off from p. K-edge-off behavior (of an input x) is essentially the observable values encountered in all k-edge-off paths (of x). Observe that for `main_cat` and `main_touch`, although the 0-edge-off behaviors (i.e., the original executions) are not distinguishable, the 1-edge-off behaviors are quite different, e.g., the behavior of `main_cat` includes those from the delegated function at line 17. However, there is a practical challenge: covering all k-edge-off behavior even when $k = 2$ may be infeasible for complex programs since the number of k-edge-off paths grows exponentially with k. Moreover, controlling the sampling process exclusively by k induces substantial noise due to code optimizations/transformations. Specifically, optimizations substantially change program structures, adding/removing predicates. The k-edge-off behaviors are hence quite different. An example can be found in Section A of an extended version of this paper [47]. To suppress the noise introduced by optimizations, we leverage the observation that optimizations rarely change the (selectivity) ranking of predicates with the maximum and minimum *dynamic selectivity.*

Definition 2.3. *Dynamic selectivity for a predicate instance $x \otimes y$ is $\left|[[y]] - [[x]]\right|$, where $[[y]]$, $[[x]]$ are the runtime values of variable x and y, and $\otimes \in \{>, \geq, \neq, ==, <, \leq\}$.*

For instance, suppose that in an execution, the value of `inbuf[i]` at line 22 in Fig. 1 is 173. It is then compared with 0x20. The dynamic selectivity of the predicate instance is hence 141 (i.e., |173 − 0x20|). Essentially, the dynamic selectivity reflects how likely a branch predict evaluates to true [37]. Although automatic code transformations may change dynamic selectivity, the predicate instances with the largest/smallest dynamic selectivity tend to stay as the largest/smallest ones after transformations. We formally explain the observation in Section 3.5 and empirically validate it in Section 4.5. Therefore, we select predicate instances to flip following a *Beta-distribution* [17] with $\alpha = \beta = 0.03$. The distribution has the largest probabilities for predicates with the minimum and maximum selectivity and small probabilities in the middle (like a U shape). Intuitively, if two programs are equivalent/similar, their predicates with the largest and the smallest selectivity tend to be the same. By flipping these predicates in the two versions, we are exploring their equivalent new behavior.

In our example, for both the optimized and the unoptimized version of `main_cat`, the algorithm first flips the predicate at line 2 with a high probability since the `-1!=c` has the largest selectivity on path 2-3-8. Then we achieve the 1-edge-off path 2-11-13-16-17-19 as discussed above. Along the new path, the algorithm flips the predicate with the largest selectivity at line 22 for further 2-edge-off exploration in both versions, exposing similar behavior.

To realize the probabilistic execution model, we develop a binary interpreter that can feed the binary with specially crafted inputs and sample observable values (Section 3.3). It also features a probabilistic memory model that can tolerate invalid memory accesses while

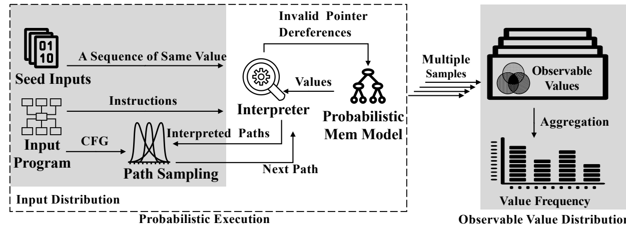


Figure 4: Workflow of PEM

ensuring equivalent observable values for equivalent programs (Section 3.6). Compared to traditional forced-execution-based techniques, PEM naturally handles the function inlining problem as our sampling is not delimited by function boundaries and our execution contexts are largely realistic. Compared to fuzzing based techniques, ours does not rely on solving the hard problem of generating valid inputs. Compared to Machine Learning based techniques, our technique focuses on dynamic behavior of programs, which are more accurate reflections of program semantics [20].

3 DESIGN

3.1 Overall Workflow

The workflow of PEM is shown in Fig. 4. The input is in the grey box on the left side. It consists of a set of seed inputs, each being an infinite sequence of the same value, the binary executable, and a path sampling strategy that can predict the next path to interpret based on the set of interpreted paths. The interpreter interprets the subject binary on a seed input, supplying the same value to any input variable encountered during interpretation, to eliminate any semantic differences caused by parameter order differences. The interpretation also strictly follows the path indicated by the path sampling component. When invalid pointer dereferences are encountered, which can be easily detected, the interpreter interacts with the probabilistic memory model to emulate the access outcomes. The emulation ensures that the same sequence of (observable) values are returned for equivalent paths. After sampling, on the right side, the observable value distributions are summarized for later similarity analysis, which simply compares two multi-sets.

The remainder of this section is organized as follows. We first model binary instructions using a simplified language. Then we present the semantic rules. After that, we discuss the path sampling method and the probabilistic memory model.

3.2 Language

The syntax of our language is in Fig. 5. A program P consists of a sequence of instructions. There are three categories of instructions. First, there are instructions that move values among registers: $r_1 = r_2$ moves the value in r_2 to r_1; $r = v$ moves a literal value v; $r_1 = r_2 \diamond r_3$ moves the result of $r_2 \diamond r_3$ to r_1. The second category is load and store instructions. The load instruction $r_1 = [r_2]$ treats the value in r_2 as a memory address and loads the value in the specified memory location to r_1. Store is similar. There are also instructions that change the control flow. Instruction **jmp** a jumps to the instruction at a; **jcc** r a performs the jump operation only when the value in r is non-zero; **jr** r is an indirect jump that uses the value in r as the target address. Instruction **done** means the interpretation is finished. Although our language does not model functions for

$$\begin{array}{l}
\langle\texttt{Program}\rangle\ P\ ::=\ I \quad \langle\texttt{Register}\rangle\ R\ ::=\ \{r_0, r_1, \ldots, r_{31}\} \\
\langle\texttt{Val}\rangle\ v\ ::=\ \{0,1,2,\ldots\} \qquad \langle\texttt{Addr}\rangle\ a\ ::=\ \{0,1,2,\ldots\} \\
\langle\texttt{Comparison}\rangle\ \otimes\ ::=\ \{==, \geq, \leq, \ldots\} \qquad \langle\texttt{BinOp}\rangle\ \diamond\ ::=\ \{+,-,*,\ldots\} \\
\langle\texttt{Instruction}\rangle\ I\ ::=\ r_1 = r_2 \mid r = v \mid r_1 = r_2 \otimes r_3 \mid r_1 = r_2 \diamond r_3 \\
\quad \mid\ r_1 = [r_2] \mid [r_1] = r_2 \mid \texttt{jmp}\ a \mid \texttt{jcc}\ r\ a \mid \texttt{jr}\ r \mid \texttt{done} \mid I_1; I_2
\end{array}$$

Figure 5: Syntax of Our Language

$$\begin{array}{ll}
Regs \in \texttt{RegState} ::= \texttt{Register} \rightarrow \texttt{Val} & \\
M \in \texttt{Memory} ::= \texttt{Addr} \rightarrow \texttt{Val} & ic \in \texttt{InstrCnt} ::= \mathbb{Z}+
\end{array}$$

$\bot$ ::= Undefined Value $p \in$ Path ::= InstrCnt $\rightarrow \mathbb{Z}+$
$s \in$ SeedValue ::= $\mathbb{Z}+$
decode(a): returns instructions in a basic block starting from a.
valid(a): if an address a is valid (pointing to allocated memory).
invalidLd(a): load a value from an invalid address a.

Figure 6: State Domains in Interpretation (top) and Auxiliary Data and Functions (bottom)

simplicity, our implementation supports the full x86 instruction set, including function invocations and returns.

3.3 Interpretation

The state domains of the interpreter are illustrated in the upper box of Fig. 6. The register state *Regs* is a mapping from a register to a value. While in our presentation values are simply non-negative integers, our implementation distinguishes bytes, words, and strings. The memory store M is a mapping from an address to a value. We use an instruction counter ic to identify each interpreted instruction along the execution path. OV denotes the observable value statistics. It is a mapping from a value to the number of its observations, that is, how many times the value appears in the current interpretation. In the lower box, we define a number of auxiliary data/structures that are immutable during interpretation and a number of helper functions used in the semantic rules. In particular, we use $\bot$ to denote an undefined value; p to denote the path to interpret, determined by the path sampling component (for a given seed value s). It is a mapping from instruction count to an instruction address. For example, a 2-edge-off path for a seed value 994 can be $\{1000 \rightarrow \text{0x804578},\ 2000 \rightarrow \text{0x80a41f}\}$. It means that the predicate instance with the instruction count 1000 ought to take the branch starting at 0x804578 when executing the binary with the seed input 994, and the instance with count 2000 should take the branch at 0x80a41f. The helper function **decode**(a) disassembles the instructions in a basic block starting at a. The function **valid**(a) determines if an address is valid. Note that since we enforce branch outcomes and use crafted inputs, the execution states may be corrupted. This function helps detect such corrupted states and seeks help from the probabilistic memory model. The function **invalidLd**(a) loads a value from an invalid address.

Part of the semantic rules are in Fig. 7. As shown at the top of Fig. 7, the state configuration is a tuple of five entries. A rule is read as follows: if the preconditions at the top are satisfied, the state transition at the bottom takes place. For example, Rule *JccGT* says that if there is a branch a' specified in p for the current instruction count ic, the conditional jump is interpreted and the continuation is I' decoded from a'.

Intuitively, given a seed value s, the interpreter initializes all registers and parameters with the same value s, and starts interpretation from the beginning (Rule *Start*). The interpretation largely follows concrete execution semantics except the following. First,

$$\boxed{\textbf{State Configuration: } \langle Regs, I, ic, M, OV\rangle}$$

$$\frac{\texttt{EntryPoint} = en \qquad \forall r, Regs[r] = s}{\langle\emptyset,\emptyset,0,\emptyset,\emptyset\rangle \rightarrow \langle Regs, \textbf{decode}(en), 1, \emptyset, \emptyset\rangle}\ \textit{Start}$$

$$\frac{p[ic] = a' \qquad I' = \textbf{decode}(a')}{\langle Regs, \texttt{jcc}\ r\ a; I, ic, M, OV\rangle \rightarrow \langle Regs, I', ic+1, M, OV\rangle}\ \textit{JccGT}$$

$$\frac{Regs[r] \neq 0 \qquad p[ic] = \bot \qquad I' = \textbf{decode}(a)}{\langle Regs, \texttt{jcc}\ r\ a; I, ic, M, OV\rangle \rightarrow \langle Regs, I', ic+1, M, OV\rangle}\ \textit{JccT}$$

$$\frac{Regs[r] = 0 \qquad p[ic] = \bot}{\langle Regs, \texttt{jcc}\ r\ a; I, ic, M, OV\rangle \rightarrow \langle Regs, I, ic+1, M, OV\rangle}\ \textit{JccF}$$

$$\frac{Regs[r_2] = a \qquad v = M[a] \qquad Regs' = Regs[r_1 \rightsquigarrow v]}{\langle Regs, r_1 = [r_2]; I, ic, M, OV\rangle \rightarrow \langle Regs', I, ic+1, M, OV\rangle}\ \textit{LdV}$$

$$\frac{Regs[r_2] = a \qquad \textbf{valid}(a) \qquad M[a] = \bot \qquad Regs' = Regs[r_1 \rightsquigarrow s]}{\langle Regs, r_1 = [r_2]; I, ic, M, OV\rangle \rightarrow \langle Regs', I, ic+1, M, OV\rangle}\ \textit{LdUd}$$

$$\frac{\begin{array}{c}Regs[r_2] = a\\ \neg\textbf{valid}(a) \qquad v = \textbf{invalidLd}(a) \qquad Regs' = Regs[r_1 \rightsquigarrow v]\end{array}}{\langle Regs, r_1 = [r_2]; I, ic, M, OV\rangle \rightarrow \langle Regs', I, ic+1, M, OV\rangle}\ \textit{LdIv}$$

Figure 7: Interpretation Rules

when it encounters a conditional jump which is indicated by the path descriptor p to take a specific branch, it takes the specified branch (Rule *JccGT*). Otherwise, it follows the normal semantics (Rules *JccT* and *JccF*). Second, when it encounters a load, if the address is valid but the memory location has not been defined, it fills it with s (Rule *LdUd*); if the address is invalid, it fetches a value from the probabilistic memory model (Rule *LdIv*); otherwise it loads a value from the memory as usual (Rule *LdV*). Here $Regs' = Regs[r_1 \rightsquigarrow v]$ means that the register state is updated by associating r_1 to v, yielding a new state $Regs'$. Store instructions are interpreted similarly. We track all dynamic memory allocations for access validity checks. Details are elided as this is standard.

We also have a set of logging rules that describe how PEM records the statistics of observable values. We record the frequencies of memory addresses accessed, values loaded/stored, control transfer targets, and predicate outcomes. Due to space limitations, details are presented in Section B of an extended version of this paper [47].

Loops and Recursion. Since our goal is to disclose semantic similarity and not to infer semantics faithful to any executions induced by real inputs, following common practice, we unroll each loop and recursive call 20 times.

3.4 Path Sampling

We present the path sampling method in Algorithm 1. It consists of two functions. Function *interpret* at line 1 interprets the input program and flips the predicates that are indicated by *path*, a mapping from instruction count to an address (Fig. 6). Specifically, during interpretation, the algorithm flips a predicate instance to an address indicated in *path* if the corresponding instruction count is met. The function returns a list of encountered predicate instances.

Function *sample* iteratively selects a predicate instance to flip (from all the interpretation results in previous steps). Variable *candidates* denote a set of candidate predicates for flipping and *budget* the number of interpretations allowed. To begin with, PEM first interprets a faithful path without altering any branch outcome. It then adds predicates in this faithful path to the candidates list (line 7).

As shown in the loop at line 8, PEM iteratively selects a predicate to flip (line 10), composes a new path with the outcome of the

Algorithm 1: Probabilistic Path Sampling

```
1  Function interpret(program, path)
       // Interprets the input program; flips predicates indicated by
          path
       // Returns predicate instances in the form (path, instruction
          count, predicate, selectivity, outcome)
2      return [(path, ic0, pr0, sel0, out0), ...]
3  Function sample(program)
4      candidates = []// Candidate branches to flip
5      budget = 400 // Number of sample rounds
6      faithful = interpret(program, ∅)
7      candidates.add(faithful)
8      while budget ≥ 0 do
9          budget = budget - 1
10         (path,ic,pr,sel,out) = select(candidates)
11         nextPath = path ∪ {ic → getBranch(pr,¬out)}
12         results = interpret(program, nextPath)
13         candidates.add(results)
```

selected predicate flipped at line 11 (function *getBranch()* acquires the target address for the true/false branch outcome of a predicate *pr*), interprets the program according to the new path (line 12), and updates the list of candidates (line 13). Note that at line 10, to select the predicate instance to flip, PEM sorts all the candidate predicates by their dynamic selectivity. Then a real number $i \in [0, 1]$ is sampled following the *probability density function* (PDF) of a Beta-distribution [17]. PEM selects a predicate that is at the i-percentile of the sorted candidates list, i.e., $selected\_pred = sorted\_candidates[i \times (len(sorted\_candidates) - 1)]$. Details can be found in Section F of an extended version of this paper [47].

3.5 Formal Analysis of Path Sampling

The effectiveness of our path sampling algorithm piggybacks on the following theorem.

THEOREM 3.1. *Assume two functionally equivalent programs P and P'. If we interpret them along two equivalent paths and collect the predicate instances during interpretation, the predicate instances with the largest (smallest) dynamic selectivity in both programs have a larger probability to match, compared to those with non-extreme selectivity.*

While optimizations (e.g., *constraint elimination* [26]) may modify predicates to simplify control flow, predicates with the smallest and largest dynamic selectivity are most resilient to optimizations, namely, their selectivity ranking hardly changes before and after optimizations. Modifications to predicates introduced by optimizations fall into two categories: *predicate elimination* and *insertion*. A predicate relocation can be considered as first removing the predicate and then adding it to another location. Specifically, compiler may eliminate a predicate if its outcome is implied by the path condition reaching the predicate. For example, it may eliminate a predicate $x > 10$ if the path condition includes $x > 20$. On the other hand, compiler may introduce new predicates to provide control flow shortcuts. Take Fig. 8 as an example. Compiler inserts a new predicate, $x < 10$, in Fig. 8b (shown in red). The modification simplifies the control flow when x is less than 10. Note that, in these cases, the dynamic selectivity of an inserted predicate will be close to the dynamic selectivity of an existing one because these inserted predicates are derived from constraints in existing predicates.

```
1  if x == 10: ...
2  else if x == 15: ...
3  else if x == 20: ...
4  ...
5  else: abort()
```

(a) Before Optimization

```
1  if x < 10: abort()
2  if x == 10: ...
3  else if x == 15: ...
4  else if x == 20: ...
5  ...
6  else: abort()
```

(b) After Optimization

Figure 8: Example of optimization that provides control flow shortcut by inserting predicates. The compiler inserts a predicate `x<10` at line 1 in Fig. 8b. When `x<10`, the execution directly goes to `abort()` without comparing with other values.

The intuition of our theorem is hence that the rankings of predicates with the smallest/largest selectivity do not depend on whether other predicates are modified. In contrast, the predicates ranked in the middle by their selectivity are more likely to have their rankings changed when predicates are removed or added by optimization.

Proof Sketch. We formalize the intuition by first reasoning about the predicates having close to the smallest dynamic selectivity. Reasoning for the largest ones is symmetric. Suppose that for each predicate, the compiler has a probability t to eliminate it and a probability q for having a predicate inserted that ranks right before it. In either case, we say the predicate is modified. The probability that a predicate is not modified is noted as $r = 1 - t - q$. We further denote as $\mathcal{P}_k$ the probability that the k-th smallest predicate is still the k-th smallest one after optimization. It is calculated by the following formula:

$$\mathcal{P}_k = r \times \sum_{i=0}^{\lfloor \frac{k-1}{2} \rfloor} \binom{k-1}{2i} \binom{2i}{i} r^{k-1-2i} t^i q^i \tag{1}$$

Intuitively, the ranking of the k-th smallest predicate is not changed by optimizations if (a) this predicate is not modified and (b) the number of predicates with a smaller dynamic selectivity does not change. In the above formula, r represents condition (a) and the second term represents condition (b). Specifically, (b) is satisfied only when the numbers of removed and inserted predicates that rank before k are equal. Here, $\binom{k-1}{2i} r^{k-1-2i}$ means an even number ($2i$) of the $k-1$ predicates with a smaller ranking are modified, and $\binom{2i}{i} t^i q^i$ means half of the modifications are removals and the other half are insertions. We visualize the distribution of $\mathcal{P}_k$ in Fig. 9 with three sets of configurations of t and q. We can see that in all setups, $\mathcal{P}_k$ monotonically decreases when k increases. □

We also conduct an empirical study to validate our theoretical analysis. The results are visualized in Section 4.5. The results show PEM has an 80-90% chance of making correct selections and exploring equivalent paths by deterministically selecting the predicates with largest/smallest dynamic selectivity.

Advantages of Probabilistic Path Sampling Over Deterministic Selection. Note that the probability of predicates with the smallest/largest selectivity having their rankings changed by optimization is not 0, although it is smaller than others. To tolerate such certainty, we employ a probabilistic approach, meaning that we follow a Beta-distribution instead of deterministically selecting the predicates with extreme selectivities for flipping. We further conduct a formal analysis to justify why the probabilistic sampling algorithm is better than the deterministic algorithm. Intuitively, by

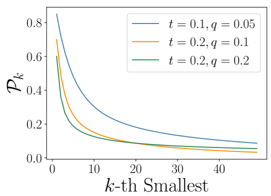


Figure 9: $\mathcal{P}_k$ w.r.t. k; The x-axis denotes the ranking of predicates by dynamic selectivity; the y-axis denotes the probability that the predicate with the k-th smallest dynamic selectivity after optimization has the same ranking. Each line shows results for one set of t and q.

following a Beta distribution, PEM spends some budget on predicates that do not have the largest or smallest selectivity, but selectivities close to the largest and smallest. These "additional" selections increase the probability that PEM selects the correct path (i.e., the equivalent path) at each step. Taking more correct steps at earlier selections increases the chance that PEM chooses a correct step at later selections because the candidate predicates of later selections come from previously explored paths. The formal proof is shown in Section C of an extended version of this paper [47].

Effect of Path Infeasibility. Our algorithm may select infeasible paths. Two possible concerns are (1) whether observable values along infeasible paths in two similar binaries can correctly disclose their semantic similarity; and (2) whether observable values along infeasible paths in two dissimilar binaries may undesirably match, leading to the wrong conclusion of their similarity.

For the first concern, we show that PEM likely selects corresponding paths when two binaries are similar, regardless of the feasibility of selected paths. That is, although the paths may be infeasible, the sequences of observable values along them are equivalent. We show a proof sketch in Section D.1 and show empirical support in Section D.3 of an extended version of this paper [47].

For the second concern, the probability that two equivalent paths are selected by PEM in two dissimilar binaries is very small. In those cases, although the initial seed paths may be undesirably similar (e.g., the error handling paths), the following flipped (infeasible) paths quickly become substantially different. The formal proof is in Section D.2 and the empirical study is in Section D.3 of an extended version of this paper [47].

3.6 Probabilistic Memory Model

The goal of the probabilistic memory model (PMM) is to handle loads and stores with invalid addresses induced by predicate flipping and the use of (out-of-bound) seed values. A key observation is that the specific values written-to/read-from the PMM do not matter as long as they can expose functional equivalence. We define the following two properties for a valid PMM.

Definition 3.1. *We say a PMM is* equivalence preserving *if the sequence of (invalid) addresses accessed, and the values written-to/read-from the PMM must be equal, for two equivalent paths in two functionally equivalent programs.*

This property ensures PEM can place equivalent programs into the same class.

Definition 3.2. *We say a PMM is* difference revealing *if the sequence of (invalid) addresses accessed, and the values written-to/read-from the PMM must be different for two different paths (pertaining invalid memory accesses) in two respective programs, which may or may not be equivalent.*

This is to ensure different programs are not mistakenly placed in the same class. For example, a naive PMM always returns a constant value for any invalid reads and ignores any invalid writes. It is equivalence preserving but not difference revealing.

Our PMM is designed as follows. Before each interpretation run, it initializes a *probabilistic memory* (*PM*), which is a mapping `Addr` $\rightarrow$ `Val` of size γ such that: $\forall a \in [0, \gamma]$, $PM[a] = random()$. An invalid memory read from the normal memory M with address a is forwarded to the PM through the **invalidLd**(a) function, which returns $PM[a \mod \gamma]$. Similarly, an invalid memory write to the normal memory M with address a and value v is achieved by setting $PM[a \mod \gamma] = v$.

It can be easily inferred that our PMM satisfies the equivalence preserving property by induction (on the length of program paths). Intuitively, the first invalid accesses in two equivalent paths must have the same invalid address. As such, our PMM must return the same random value. This same random value may be used to compute other identical (invalid) addresses in the two paths such that the following invalid loads/stores are equivalent. It also probabilistically satisfies the difference revealing property. Specifically, different paths manifest themselves by some different invalid addresses, and our PMM likely returns different (random) values for these different addresses, rendering the following memory behaviors (with invalid addresses) different. The chance that different paths may exhibit the same behavior depends on γ. Due to the complexity of modeling memory behavior in real-world program paths, we did not derive a theoretical probabilistic bound for our PMM. However, empirically we find that $\gamma = 64k$ enables very good results (with our loop unrolling bound 20). An example can be found in Section E of an extended version of this paper [47].

4 EVALUATION

We implement PEM on QEMU [35]. Details are in Section F of an extended version of this paper [47]. We evaluate PEM via the following research questions:

RQ1: How does PEM perform compared to the baselines?
RQ2: How useful is PEM in real-world applications?
RQ3: Is PEM generalizable?
RQ4: How does each component affect the performance?

4.1 Setup

We conduct the experiments on a server with a 24-core Intel(R) Xeon(R) 4214R CPU at 2.40GHz, 188G memory, and Ubuntu 18.04.

Datasets. We use two datasets. *Dataset-I*: To compare with IMF and BLEX, which only use Coreutils [8] as their dataset, we construct a dataset from Coreutils-8.32. We compile the dataset using GCC-9.4 and Clang-12, with 3 optimization levels (i.e., -O0, -O2, and -O3). *Dataset-II* includes 9 real-world projects commonly-used in binary similarity analysis projects [22, 28, 31]. They are Coreutils, Curl, Diffutils, Findutils, OpenSSL, GMP, SQLite, ImageMagick, and Zlib. The binaries are obtained from [31]. In total, we have 30 programs with 35k functions, compiled with 3 different options. Details can be found in Table 8 of an extended version of this paper [47].

Baseline Tools. We compare with 6 baselines. For execution-based methods (Baseline-I), we use IMF [43] and BLEX [12], which are SOTAs as far as we know. For Deep Learning methods (Baseline-II),

Table 1: Comparison of PEM**, IMF, and BLEX. *C* and *G* denote Clang and GCC, respectively. Each precision is averaged over the 106 binaries in Coreutils.**

Pair	Precision@1			Precision@3		Precision@5	
	PEM	IMF	BLEX	PEM	IMF	PEM	IMF
C-O0 ***C***-O3	94.5	77.5	X	98.2	84.2	98.7	86.4
C-O2 ***C***-O3	99.8	97.3	X	100.0	99.3	100.0	99.4
C-O0 ***G***-O3	**94.5**	60.1	X	97.3	70.6	98.6	73.4
G-O0 ***G***-O3	96.3	70.4	61.1	98.0	81.3	98.7	84.6
G-O2 ***G***-O3	98.6	89.5	77.1	99.4	95.5	99.8	96.2
G-O0 ***C***-O3	**92.2**	66.0	X	94.1	76.1	96.2	80.0
Average	**96.0**	**76.8**	**69.1**	**97.8**	**84.5**	**98.7**	**86.7**

we use SAFE [30] and Trex [31]. We use their pre-trained models or train using their released implementation with the default hyper-parameters. Also, we compare with the best two models (i.e., GNN and GMN) in How-Solve [28] that conducts a measurement study on Machine Learning methods.

Metrics. Following the same experiment setup in IMF and BLEX, for a function compiled with a higher level optimization option (e.g., -O3), we query the most similar function in all the functions (in the same binary) compiled with a lower level optimization option. As such, there is only one matched function. We hence use Precision at Position 1 (PR@1) as the metric. Given a function, PR@1 measures whether the matched function scores the highest out of the pool of candidate functions. Many data-driven methods [28, 30, 31, 52] use *Area Under Curve* (AUC) of the *Receiver Operating Characteristic* (ROC) curve. Existing literature [2] points out that a good AUC score does not necessarily imply good performance on an imbalanced dataset (e.g., class 1 having 1 sample and class 2 having 100). Therefore we choose PR@1 as our metric, which aligns better with the real-world (imbalanced) use scenario of binary similarity.

4.2 RQ1: Comparison to Baselines

Comparison to Baseline-I. We compare PEM with Baseline-I on Dataset-I. To conduct the evaluation, we first use PEM to sample each function in these binaries and aggregate the distribution of observable values. Then, for each function in an optimized binary, we compute its similarity score against all functions of the same program compiled with a lower optimization level, and use the ones with the highest scores to compute PR@1. Besides PR@1, we also use PR@3 and @5 for a more thorough comparison with IMF. The comparison results with IMF and BLEX are shown in Table 1[1]. The first two columns list the compilers and the optimization flags used to generate the reference and query binaries. Columns 3-5, 6-7, and 8-9 list PR@1, @3, and @5, respectively. Note that BLEX only reports PR@1 and does not have results for binaries compiled with Clang. PEM outperforms BLEX on PR@1 and outperforms IMF on all 3 metrics under all settings. Especially, for function pairs (Clang-O0, GCC-O3) and (GCC-O0, Clang-O3), which are the most challenging settings in our experiment, PEM outperforms IMF by about 25%.

Comparison to Baseline-II. We compare PEM with Baseline-II on Dataset-II. Following the setup of How-Solve [28], for each positive pair (of functions), namely, similar functions, 100 negative pairs (i.e., dissimilar functions) are introduced to build up the test set. The results are shown in Fig. 10. The x axis represents different programs, and the y axis is PR@1. The results of PEM, GNN, and

[1]We compare PEM with the reported results in the IMF paper and contact the authors of IMF to ensure our setups are the same.

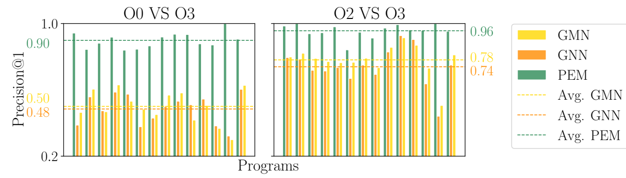


Figure 10: Comparison with How-Solve. We leverage the best two models (i.e., GNN and GMN) in How-Solve. Each bar denotes a program, whose name is elided. A bar with 1.0 PR@1 means that PEM **finds the correct matches for all functions in the program. Dashed lines denote the average PR@1 of each tool.**

GMN are shown in green, yellow, and red bars. The average PR@1 of each tool is marked by the dashed line with the related color. Note that GNN and GMN are the best two models out of all 10 ML-based methods in How-Solve [28] (including Trex and SAFE). As Fig. 10 illustrates, PEM achieves scores from 0.84 to 1.00, which is around 20-40% better than GNN and GMN.

Comparison with Trex and SAFE. With the aforementioned composition of dataset, PEM outperforms Trex by 40% and outperforms SAFE by 25% on average. Moreover, the performance of Trex and SAFE is sensitive to dataset composition. Hence in this comparative experiment, we analyze how different data compositions affect the performance of different tools. Our results show that PEM is 50% more resilient than Trex and SAFE. Details can be found in Section H of an extended version of this paper [47].

4.3 RQ2: Real-World Case Study

We demonstrate the practice use of PEM via a case study of detecting 1-day vulnerabilities. Suppose that after a vulnerability is reported, a system maintainer wants to know if the vulnerable function occurs in a production system. She can use PEM to search for the vulnerable function from a large number of binary functions and decide whether further actions should be taken (e.g., patch the system). We collect 8 1-day Vulnerabilities (CVEs) and use the optimized version of the problematic function to search for its counterpart in the unoptimized binary. The results show that in 7 out of the 8 cases, our tool can find the ground truth function as the top one, while the other two ML-based methods each can only find 1 of them. Even if we look into the top 30, both of them can only find 2 of these problematic functions. Details can be found in Section I of an extended version of this paper [47].

4.4 RQ3: Generalizability

We evaluate the generalizability of PEM from three perspectives. First, we show that PEM is efficient so that it can scale to large projects. Second, we illustrate that PEM has good code coverage for most functions. That means it can explore enough semantic behavior even for complex functions. Last but not least, besides x86-64, we show that PEM can support another architecture with reasonable human efforts, meaning that PEM can be easily generalized to analyzing binary programs from multiple architectures, without the need of substantial efforts in building lifting or reverse engineering tools to recover high-level semantics from binaries.

Efficiency. PEM analyzes more than 3 functions per second in most cases. Note that this is a one-time effort. After interpretation and generating semantic representations, PEM searches these representations to find similar functions. PEM compares more than 2000

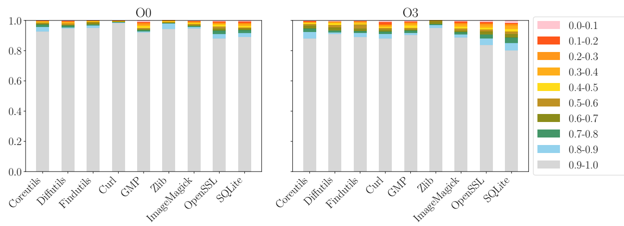


Figure 11: Coverage of PEM

pairs per second in most cases. The comparison can be parallelized. With 4 processes, we are able to compare 1.7 million function pairs in 4 minutes (wall-clock). We visualize the results in Figure 27 of an extended version of this paper [47].

PEM takes 13 minutes to cover more than 95% code for all functions in Coreutils (with a single thread). In comparison, the forced-execution based method BLEX takes 1.2 hours. In our experiment, PEM takes 26 minutes to process two Coreutils binaries compiled with different optimization levels, and it takes another 14 minutes to compare all 1.7 million function pairs between these two binaries, yielding a total time cost of 40 minutes. While IMF takes 32 minutes to complete the same task, PEM achieves significantly better precision than IMF. Machine learning models typically have an expensive training time. They have better performance in test time.

Coverage. The code coverage of PEM on Dataset-II is shown in Fig. 11. The x axis marks the projects and the y axis shows the percentage of functions for which PEM has achieved various levels of coverage, denoted by different colors. As we can see, 90% of the functions in -O0 and 85% functions in -O3 have a full or close-to-full coverage. Those functions with less than 40% coverage have extremely complex control flow structures, with many inlined callees. For example, the main function of *sort* in Coreutils has 496 basic blocks, resulting in millions of potential paths. Note that even with such a huge path space, PEM is still able to select similar paths and collect consistent values with a high probability.

Cross-arch Support. We add AArch64 [1] support to PEM with only around 200 lines of C++ code and 0.5 person-day efforts. This is possible because our probabilistic execution model is general and does not rely on specialized features from the underlying architecture. PEM achieves a PR@1 of 86.8 for Coreutils (-O0 and -O3) on AArch64, whereas its counterpart on x86-64 is 89.4. In addition, it achieves a PR@1 of 84.9 when we query with functions compiled on x86-64 in the pool of functions compiled on AArch64. Details can be found in Table 7 of an extended version of this paper [47].

4.5 RQ4: Ablation Study

Probabilistic Path Sampling. First, we empirically validate our hypothesis that branches with the largest and smallest selectivity are stable before and after code transformations. We collect equivalent interpretation traces from the main functions in Coreutils binaries compiled with different options. Then we analyze the matching traces and check if the predicates with the largest and the smallest selectivity in these cross-version traces match, leveraging the debug information. In total, we study 636 traces from 6 binaries with a total of 16k predicate instances. We observe that with a probability of more than 80%, our hypothesis holds. The detailed results are shown in Fig. 12. From the two ends of the lines, we can observe that in more than 80% cases, the predicates with the smallest and the largest selectivity match. In contrast, those in the middle do not have such a property. The median for the *max-3* selectivity is even close to 0%. For predicate instances with the smallest/largest selectivity in one trace (e.g., -O3), we further study the selectivity rankings of their correspondences in the other trace (e.g., -O0). The results are visualized in Fig. 13. Observe that in more than 98% cases, they have the top-3 smallest or largest selectivity in the other trace.

Table 2: Perf. w.r.t. Different Path Sampling Strategies

	LastPred	Det.	PEM
PR@1	40.24	79.27	**91.46**
Cover-O0	66.28	96.77	96.81
Cover-O3	53.14	92.95	92.97

Table 3: Perf. w.r.t. Different Budgets

	1	20	50	100	200	400	600
PR@1	70	74	79	81	85	86	86
Cover-O0	63	87	91	94	96	98	98
Cover-O3	51	79	84	88	93	95	96

Table 4: Perf. w.r.t. Different Memory Models

	No-Mem	Const	PMM
PR@1	76.35	83.48	**85.75**
Cover-O0	97.59	97.70	**98.03**
Cover-O3	94.39	95.11	**95.52**

Furthermore, we select 80 most challenging functions in Coreutils to further study the effectiveness of our path sampling strategy. These functions have more than 150 basic blocks and the average connectivity is larger than 3, namely, a block is connected to more than 3 blocks on average. We compare the performance of 3 path sampling strategies. The results are shown in Table 2. The three rows show the PR@1, the code coverage for -O0 and -O3 functions, respectively. The second column presents a strategy in which PEM flips the last predicate encountered in the previous round with an uncovered branch. The third column denotes a strategy in which PEM deterministically flips the predicates with the largest and the smallest selectivity at each round. The last column presents our probabilistic path sampling strategy. Observe that the probabilistic strategy substantially outperforms the other two and both the deterministic and probabilistic strategies can achieve good coverage.

Code Coverage versus Precision. We run PEM with different round budgets on Coreutils and observe coverage and precision changes. The results are shown in Table 3. Observe that if we only interpret each function once without any flipping, the precision is as low as 70 and the coverage is low too. With more budgets, namely, flipping more predicates, both the precision and the coverage improve, indicating PEM can expose equivalent semantics. But the improvement becomes marginal after 200.

Probabilistic Memory Model (PMM). We run PEM with different memory model setups on Coreutils to illustrate the benefit of modeling invalid memory accesses. The results are in Table 4. Specifically, *No-Mem* means we do not model invalid memory accesses. We return random values for invalid reads and simply discard invalid writes. The precision of No-Mem is nearly 10% lower than PMM, while their coverage is similar. That is because some dependencies between memory accesses are missing without handling invalid writes. On the other hand, if we allow writes to invalid memory regions but always return a constant value for all invalid reads, as shown in the column of *Const*, the precision is better than No-Mem. However, it

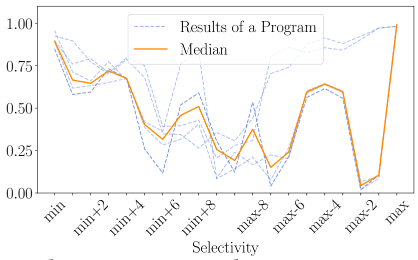


Figure 12: Predicate Correspondence versus Dynamic Selectivity. Each blue dashed line represents the analysis results of path pairs from two respective binaries compiled differently from a program. The x axis represents selectivity (with *min* the minimal and *max* the maximum) and y denotes the percentage of predicate matches. We also compute the median for each selectivity, resulting in the orange line.

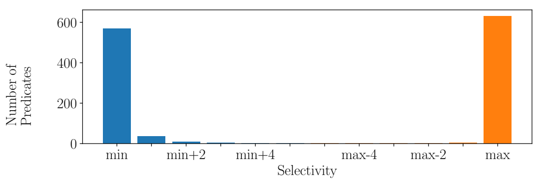


Figure 13: Correspondence of Predicates with Min and Max Selectivity. Blue is for min and orange for max. For example, the bar at min+1 means that about 20 predicates with min selectivity in one trace have min+1 selectivity in the other trace.

is still inferior to PMM. This is due to returning the constant value making reads from different invalid addresses indistinguishable.

Robustness. We alter system configurations of PEM and run random sampling for each probabilistic component in PEM. The experimental results show that PEM is robust with regard to different configurations and variances in samplings. Details can be found in Section G of an extended version of this paper [47].

5 RELATED WORK

Binary Similarity. Many existing techniques aim to detect semantically similar functions, driven by static [11, 24] and dynamic [12, 15, 16, 43] analysis. A number of representative methods have been discussed in Section 2.2. Other techniques compare code similarity at different granularity, e.g., whole binary [25, 48], assembly [10, 13, 45], and basic block [33]. While our method represents semantics at the function level, the resulting value sets of our system can be used as function semantic signatures and facilitate comparisons working at other granularity.

Forced Execution. Forced execution [12, 32, 51, 55] concretely executes a binary along different paths by flipping branch outcomes. They typically aim to cover more code in a program and thus use coverage as the guidance. They can hardly select similar sets of paths for the same program compiled with different optimizations. Their focus is on recovering from invalid memory accesses. In contrast, the probabilistic memory model of PEM reveals the different semantics introduced by different invalid accesses with high probability.

6 CONCLUSION

We develop a novel probabilistic execution model for effective sampling and representation of binary program semantics. It features a path-sampling algorithm that is resilient to code transformations and a probabilistic memory model that can tolerate invalid memory accesses. It substantially outperforms the state of the arts.

7 DATA AVAILABILITY

Our experimental data and the artifact are available at [46].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and suggestions. This research was supported, in part by DARPA VSPELLS - HR001120S0058, IARPA TrojAI W911NF-19-S-0012, NSF 1901242 and 1910300, ONR N000141712045, N000141410468 and N000141712947. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] ARM64 2022. *Learn the architecture - AArch64 Instruction Set Architecture.* https://developer.arm.com/documentation/102374/latest/
[2] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2020. Dos and Don'ts of Machine Learning in Computer Security. *CoRR* abs/2010.09470 (2020). arXiv:2010.09470 https://arxiv.org/abs/2010.09470
[3] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*. 845–860.
[4] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM.
[5] BinDiff 2022. *zynamics BinDiff.* https://www.zynamics.com/bindiff.html
[6] Dong-Kyu Chae, Jiwoon Ha, Sang-Wook Kim, BooJoong Kang, and Eul Gyu Im. 2013. Software plagiarism detection: a graph-based approach. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 1577–1580.
[7] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350
[8] Coreutils 2022. *Coreutils - GNU core utilities.* https://www.gnu.org/software/coreutils/
[9] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 266–280. https://doi.org/10.1145/2908080.2908126
[10] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. 472–489. https://doi.org/10.1109/SP.2019.00003
[11] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. https://doi.org/10.14722/ndss.2020.24311
[12] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 303–317.
[13] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 480–491.
[14] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE.
[15] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. *VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary.* Association for Computing Machinery, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480
[16] Y. Hu, Y. Zhang, J. Li, H. Wang, B. Li, and D. Gu. 2018. BinMatch: A Semantics-Based Hybrid Approach on Binary Code Clone Analysis. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer

Society, Los Alamitos, CA, USA, 104–114. https://doi.org/10.1109/ICSME.2018.00019

[17] N. L. Johnson and S. Kotz. 1972. *Distributions in Statistics: Continuous Multivariate Distributions.* John Wiley, New York, NY.

[18] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14).*

[19] Chariton Karamitas and Athanasios Kehagias. 2018. Efficient features for function matching between binary executables. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).* 335–345. https://doi.org/10.1109/SANER.2018.8330221

[20] Ulf Kargén and Nahid Shahmehri. 2017. Towards robust instruction-level trace alignment of binary code. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* 342–352. https://doi.org/10.1109/ASE.2017.8115647

[21] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer.

[22] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *IEEE Transactions on Software Engineering* (2022), 1–23. https://doi.org/10.1109/TSE.2022.3187689

[23] Geunwoo Kim, Sanghyun Hong, Michael Franz, and Dokyung Song. 2022. Improving Cross-Platform Binary Analysis Using Representation Learning via Graph Alignment. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022).* Association for Computing Machinery, New York, NY, USA, 151–163. https://doi.org/10.1145/3533767.3534383

[24] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. 2019. Graph Matching Networks for Learning the Similarity of Graph Structured Objects. *CoRR* abs/1904.12787 (2019). arXiv:1904.12787 http://arxiv.org/abs/1904.12787

[25] 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.* Association for Computing Machinery, New York, NY, USA, 667–678. https://doi.org/10.1145/3238147.3238199

[26] LLVM 2022. *llvm-project.* https://github.com/llvm/llvm-project/blob/release/12.x/llvm/lib/Transforms/Scalar/ConstraintElimination.cpp

[27] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2017. Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection. *IEEE Transactions on Software Engineering* 43, 12 (2017), 1157–1177. https://doi.org/10.1109/TSE.2017.2655046

[28] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *USENIX 2022, 31st USENIX Security Symposium, 10-12 August 2022, Boston, MA, USA,* Usenix (Ed.). Boston.

[29] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. *CoRR* abs/2103.11626 (2021). arXiv:2103.11626 https://arxiv.org/abs/2103.11626

[30] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2018. SAFE: Self-Attentive Function Embeddings for Binary Similarity. https://doi.org/10.48550/ARXIV.1811.05296

[31] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. https://doi.org/10.48550/ARXIV.2012.08680

[32] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14).* USENIX Association, USA, 829–844.

[33] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *2015 IEEE Symposium on Security and Privacy.* 709–724. https://doi.org/10.1109/SP.2015.49

[34] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '14).* Association for Computing Machinery, New York, NY, USA, 406–415. https://doi.org/10.1145/2664243.2664269

[35] QEMU 2023. *A generic and open source machine emulator and virtualizer.* https://www.qemu.org

[36] 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.* Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035

[37] Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PREACH: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE).* 1706–1717. https://doi.org/10.1145/3510003.3510227

[38] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE).* 1157–1168. https://doi.org/10.1145/2884781.2884877

[39] Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. *ACM Trans. Softw. Eng. Methodol.* 30, 1, Article 6 (dec 2021), 36 pages. https://doi.org/10.1145/3412376

[40] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15).* 611–626.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[42] Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *SIGAPP Appl. Comput. Rev.* 19, 4 (jan 2020), 28–39. https://doi.org/10.1145/3381307.3381310

[43] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017).* IEEE Press, 319–330.

[44] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023).* Association for Computing Machinery, New York, NY, USA, 1106–1118. https://doi.org/10.1145/3597926.3598121

[45] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM. https://doi.org/10.1145/3133956.3134018

[46] Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. *PEM.* https://github.com/XZ-X/PEM.git

[47] Xiangzhe Xu, Zhou Xuan, Shiwei Feng, Siyuan Cheng, Yapeng Ye, Qingkai Shi, Guanhong Tao, Le Yu, Zhuo Zhang, and Xiangyu Zhang. 2023. PEM: Representing Binary Program Semantics for Similarity Analysis via a Probabilistic Execution Model. arXiv:2308.15449 [cs.SE]

[48] Xi Xu, Qinghua Zheng, Ming Fan, Jia Ang, and Ting Liu. 2021. Interpretation-enabled Software Reuse Detection Based on a Multi-Level Birthmark Model. 873–884. https://doi.org/10.1109/ICSE43902.2021.00084

[49] Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu, and Fu Song. 2017. SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* 462–472. https://doi.org/10.1109/ICSE.2017.49

[50] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2022. Codee: A Tensor Embedding Scheme for Binary Code Search. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2224–2244. https://doi.org/10.1109/TSE.2021.3056139

[51] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. PMP: Cost-effective Forced Execution with Probabilistic Memory Pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP).* 1121–1138. https://doi.org/10.1109/SP40000.2020.00035

[52] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order Matters: Semantic-Aware Neural Networks for Binary Code Similarity Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 01 (Apr. 2020), 1145–1152. https://doi.org/10.1609/aaai.v34i01.5466

[53] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating Faults through Automated Predicate Switching. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06).* Association for Computing Machinery, New York, NY, USA, 272–281. https://doi.org/10.1145/1134285.1134324

[54] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-Chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. OSPREY: Recovery of Variable and Data Structure via Probabilistic Analysis for Stripped Binary. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021.* IEEE, 813–832. https://doi.org/10.1109/SP40001.2021.00051

[55] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 137:1–137:31. https://doi.org/10.1145/3360563

Received 2023-02-02; accepted 2023-07-27