Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis

Xiangzhe Xu
Purdue University
West Lafayette, USA
xu1415@purdue.edu

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

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

Guangyu Shen
Purdue University
West Lafayette, USA
shen447@purdue.edu

Zian Su
Purdue University
West Lafayette, USA
su284@purdue.edu

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

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

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

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

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

ABSTRACT

Given a function in the binary executable form, binary code similarity analysis determines a set of similar functions from a large pool of candidate functions. These similar functions are usually compiled from the same source code with different compilation setups. Such analysis has a large number of applications, such as malware detection, code clone detection, and automatic software patching. The state-of-the art methods utilize complex Deep Learning models such as Transformer models. We observe that these models suffer from undesirable instruction distribution biases caused by specific compiler conventions. We develop a novel technique to detect such biases and repair them by removing the corresponding instructions from the dataset and finetuning the models. This entails synergy between Deep Learning model analysis and program analysis. Our results show that we can substantially improve the state-of-the-art models' performance by up to 14.4% in the most challenging cases where test data may be out of the distributions of training data.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; • **Computing methodologies → Machine learning**.

KEYWORDS

Binary Similarity Analysis, Transformer, Program Analysis

ACM Reference Format:
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 (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597926.3598121

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598121

1 INTRODUCTION

Given a query function f, *code similarity analysis* [1, 5, 16, 28, 29] finds the functions from a pool P that are similar to f. Similarity analysis has a wide range of applications in automatic software patching [3, 36, 46, 55, 56, 68], software plagiarism detection [9, 33, 52, 60, 71], and 1-day vulnerability detection [15, 62]. For instance, a critical security vulnerability might be detected in a library. It is essential to detect whether an existing project contains the problematic function from that library. *Binary similarity analysis* is a special kind of code similarity analysis. It handles functions in the executable form, without source code or any symbolic information. In the context of binary similarity analysis, similar functions are usually compiled from the same source code with different compilers or compilation options. It is particularly useful in reverse engineering [21, 22, 44, 62] and malware analysis [6, 10, 18, 19, 24, 26, 70]. Traditionally, binary similarity analysis is achieved using classic program analysis such as control-flow differential analysis [7, 25], program dependence analysis [67], symbolic execution [13, 33], and trace analysis [13, 17, 22, 40, 62].

Recent research has shown that *Deep Learning models*, such as Transformer models, can achieve state-of-the-art results in binary similarity analysis, outperforming classic methods [44, 61]. For example, JTrans [61] uses Transformer models and achieves 62.5% accuracy, 30.5% better than the prior work. In JTrans, the model first encodes f and every function in P to embeddings. It then computes the cosine similarity between the embedding of f and the embeddings of all functions in P. A high cosine similarity value between a pair of embeddings indicates that the model considers the related two functions similar. Functions in P are further ranked by the similarity values and the top-k functions may be selected as similar functions. The model is trained on a large set of function pairs that are labeled as *similar* or *dissimilar*, using contrastive learning. In total, there are over 3 million function pairs used in training, much more than a few other similar methods [27, 37, 44].

However, as we will show in Section 2.2, the inherent compiler conventions (e.g., generating specific instruction sequences at function prologues) introduce instruction distribution biases, some of which are undesirably associated with specific similarity analysis results. For example, a particular instruction appearing in the prologue of a function but not in the other similar function may lead to a misclassification of dissimilarity. However, we cannot simply remove instruction distribution biases because many of them are not caused by compilers, but rather due to unique program semantics. In other words, they may be important to correct classification.

In this paper, we aim to identify and suppress undesirable biases to improve Transformer-based binary similarity analysis models. In particular, we first determine a set of instructions that are important to classification results (regarding all training samples). We then use program analysis to determine if these instructions indeed have semantics importance. If not, they are likely due to compiler introduced biases. We further propose to preclude these instructions by removing them from all the binary functions and finetuning the model with the updated dataset.

Our contributions are summarized as follows.

- We propose a novel instruction deemphasis technique that can effectively prevent a model from learning instruction distribution biases introduced by compilers, and improve the generalizability of models on out-of-distribution data.
- We devise a method to identify instructions from a binary function that may significantly affect the classification results of a binary similarity analysis model.
- We propose a novel metric that can measure the importance of a single instruction to the semantics of a binary function.
- We develop a prototype DiEmph. We conduct experiments on 6 state-of-the-art models and evaluate the effectiveness of DiEmph on 7 real-world projects widely-used for the binary similarity tasks. The results show that DiEmph improves models' performance on out-of-distribution data by 3.7–14.4%. In the most practical application scenario, namely, using the most complex model trained on the largest dataset, DiEmph achieves 14.4% accuracy improvement on out-of-distribution data, from 37.2% to 51.6%.

2 MOTIVATION

In this section, we use an example to discuss the limitation of existing methods and illustrate our method.

2.1 Motivating Example

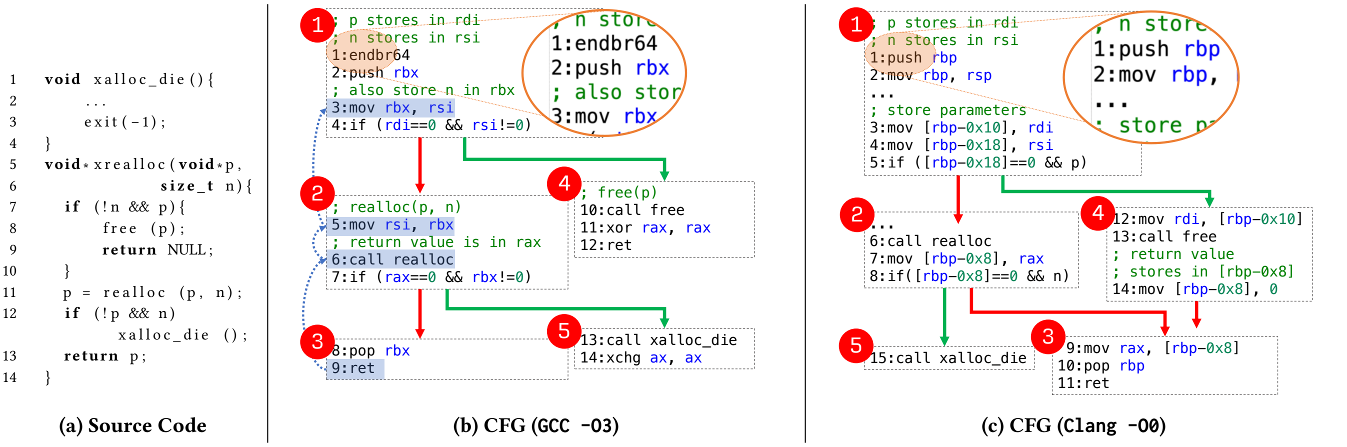
The example is simplified from the function `xrealloc()` in Coreutils [12]. It is shown in Fig. 1a. The function is used to change the sizes of an allocated memory region. It takes as input two parameters. The first one is a pointer to an allocated memory region, and the second is a new size. In common cases, the function returns a pointer with the new size (at line 13). If the second parameter is zero, `xrealloc` will free the region and return a null pointer (at line 9). The *if*-statement at line 12 handles exceptional cases: if the function `realloc` returns a null pointer, a non-return error processing function `xrealloc_die` will be invoked. The function emits an error message and terminates the execution.

We compile the function with the GCC compiler and the option `-O3`, and with the Clang compiler and the option `-O0`. The resulting control flow graphs (CFGs) are shown in Fig. 1b and Fig. 1c, respectively. We number the basic blocks and list the numbers in the red circle at the upper left corner of each basic block. Note that we label the corresponding basic blocks with the same numbers. At the beginning of both CFGs, the two parameters `p` and `n` are stored in registers `rdi` and `rsi`, respectively. The first basic blocks of both functions contain the function prologue (e.g., saving registers and allocating a stackframe). Basic blocks 2–3 contain the program logic for common cases: invoking `realloc` and returning its result. Basic blocks 4 (in both CFGs) free the pointer `p` when the new size `n` is zero. Basic blocks 5 process exceptional cases as shown at line 12 of Fig. 1a.

2.2 Limitations in State-of-the-Art Models

The CFGs in Fig. 1b and Fig. 1c have similar control structures and have many corresponding instructions. However, the state-of-the-art Transformer-based method JTrans [61] produces a low similarity score (0.3) to this pair of CFGs. The majority of similar function pairs are expected to have a score larger than 0.5. Thus the model mistakenly concludes these two CFGs are not similar. We investigate the results and find that the model is undesirably sensitive to some special patterns in the function prologue, namely, these patterns are considered very important to classification results by the model. Some of the misclassification-inducing patterns are highlighted by the orange circles in Fig. 1b and Fig. 1c. Specifically, in the optimized version (Fig. 1b), the GCC compiler inserts an additional instruction `endbr64` at the beginning. The instruction is irrelevant to the program functionality. It is used to support *control flow integrity* [54], a security feature on recent processors. However, the model tends to consider a pair of functions likely dissimilar when one has the instruction and the other does not. By manually adding the `endbr64` instruction to the un-optimized version, we can increase the similarity score for this pair of binary functions to 0.60. Even removing the instruction from the optimized version allows us to improve it to 0.52.

We further investigate the training dataset and find that the `endbr64` instruction has an undesirable bias in distribution. The training data of JTrans [61] contains over 3 million binary functions with different functionalities. For each pair of binary functions in the training dataset, there are two possible labels, i.e., *similar* and *dissimilar*, whose ratio is 1:2. Following the same ratio, we randomly select 0.9 million pairs of similar functions and 1.8 million pairs of dissimilar functions and study the distribution of the `endbr64` instruction in these two kinds of function pairs. The results are shown in Fig. 2. Observe that when only one of the function has the instruction, the numbers of similar and dissimilar function pairs have the ratio of 1:27, substantially deviated from the 1:2 ratio. Hence the trained model undesirably associates the inconsistency of `endbr64`'s presence with the conclusion of dissimilarity. Note that not all distribution biases are problematic. It is quite common that certain instructions (or sequences) are representative for specific functionalities. Therefore, a simple idea of removing all biases can hardly work.



```
void xalloc_die(){
    ...
    exit(-1);
}
void* xrealloc(void*p,
               size_t n){
  if (!n && p){
      free (p);
      return NULL;
  }
  p = realloc (p, n);
  if (!p && n)
         xalloc_die ();
  return p;
}
```

(a) Source Code

```
; p stores in rdi
; n stores in rsi
1:endbr64
2:push rbx
; also store n in rbx
3:mov rbx, rsi
4:if (rdi==0 && rsi!=0)

; realloc(p, n)
5:mov rsi, rbx
; return value is in rax
6:call realloc
7:if (rax==0 && rbx!=0)

8:pop rbx
9:ret

; free(p)
10:call free
11:xor rax, rax
12:ret

13:call xalloc_die
14:xchg ax, ax
```

(b) CFG (GCC -O3)

```
; p stores in rdi
; n stores in rsi
1:push rbp
2:mov rbp, rsp
...
; store parameters
3:mov [rbp-0x10], rdi
4:mov [rbp-0x18], rsi
5:if ([rbp-0x18]==0 && p)

...
6:call realloc
7:mov [rbp-0x8], rax
8:if([rbp-0x8]==0 && n)

 9:mov rax, [rbp-0x8]
10:pop rbp
11:ret

12:mov rdi, [rbp-0x10]
13:call free
; return value
; stores in [rbp-0x8]
14:mov [rbp-0x8], 0

15:call xalloc_die
```

(c) CFG (Clang -O0)

Figure 1: Motivating example. The two control flow graphs (CFGs) are the compilation results of Fig. 1a. Green and red edges in the CFGs denote the control flow that the related branch is taken, or not taken, respectively. Red edges also denote the default control flow. We number the basic blocks and show the numbers in red circles.

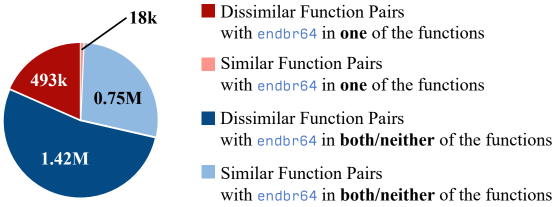


Figure 2: Undesirable bias in the distribution of training dataset. We divide samples in the training dataset of JTrans into four categories. The pie chart depicts the number of samples in each category.

2.3 Our Technique

Existing techniques suffer from biased distributions of certain instructions. These biases are caused by compilers whose inherent behaviors may not strictly follow a normal distribution due to their deterministic nature. The overarching idea of our technique is to deemphasize instructions that do not denote essential program semantics, such as the aforementioned endbr64 instruction added to support control-flow-integrity.

We determine instructions that are important to classification results over the entire training set. The importance of an instruction *i* is determined by embedding changes of all functions involving *i* when we remove *i* from the function (Section 3.1). It is also called the *classification importance*. For an instruction *i* of top-*k* classification importance, we use program analysis to determine *i*'s *semantics importance*. Specifically, given a function, if *i* is in the backward slice of any *stable variable*, (e.g., a heap variable, or a return variable) in the function, *i* is considered semantically important to the function. A variable is unstable if it is likely changed by compiler optimization. A local variable of a primitive type (e.g., int) is unstable because it may be placed on stack in one version and in a register in another (optimized) version. Stable variables tend to have consistent representations in similar functions. Intuitively, an instruction is considered important if it directly or transitively contributes to the computation of some stable variables. We say *i* is semantically important regarding a dataset if it is important for at least a certain number of functions in the dataset. Additional challenges need to be addressed in the backward slicing as certain dependences need to be precluded (Section 3.2). If an instruction has top classification importance but not semantics importance, it is removed from all the functions in the training dataset. We then finetune the model using the reduced dataset to repair the undesirable biases.

In our example, the endbr64 at the beginning of Fig. 1b has a relatively high classification importance of 0.1, while the instruction call realloc has an importance of only 0.05. That is, the model considers endbr64 even more important than the call instruction. Although endbr64 has classification importance, it is not semantically important. In particular, we identify all the stable variables in the function and then determine the important instructions. For instance, the ret instruction (at line 9 in block 3) implicitly returns the value in rax to the caller[1]. Thus rax@9 (meaning rax at line 9) is a stable variable. We highlight all the instructions that affect rax@9 with blue shadow (i.e., its backward slice). The blue dashed arrows denote the dependences. First, rax@9 is (implicitly) defined by the function call realloc at line 6 through its return. And the result of the function call is affected by its parameters. The first parameter (rdi) is defined at the function entry, and the second one (rsi) is defined at line 5. The instruction mov rsi, rbx means that copying the value in register rbx to rsi. The variable rsi@5 hence depends on rbx@5, which is defined at line 3. The variable rbx@3 is in turn defined by rsi from the function entry. Thus our analysis identifies that instructions at lines 3, 5, 6 affect the variable rax@9 and consider them important. The endbr64 instruction is considered semantically unimportant.

Removing endbr64 and a few more instructions of similar nature from all functions in the dataset and fine-tuning the model allows us to improve JTrans's accuracy from 37.2% to 51.6%.

[1] At the binary level, function return value is in rax by default.

3 DESIGN

The overall workflow of DiEmph is shown in Fig. 3. At the top we show a typical training pipeline of Transformer model. The model is first pretrained. The pretraining usually consumes tremendous time and computing resources. Thus developers for downstream tasks do not need to repeat the pretraining process. Instead, they directly finetune a pretrained model on a domain-specific dataset (e.g., a binary similarity dataset in our scenario), which takes significantly less effort.

DiEmph takes as input a finetuned binary similarity model and the dataset for finetune. It aims to produce a new model with better performance via removing certain instructions from the dataset and rerunning the finetune process. Specifically, the technique samples N functions from the training dataset, and analyzes classification importance for all instructions in these N functions (Step 1). After that, it selects the instructions with exceptionally large classification importance (Step 2) and uses program analysis to determine whether these instructions are semantically important (Step 3). For instructions that have high classification importance but are not semantically important, we remove them from the training dataset (Step 4) and rerun the finetune process (Step 5). Finally, these problematic instructions will also be removed from the model input at inference time (Step 6) to ensure input space consistency in finetune and testing.

This section is organized as follows. In Section 3.1, we introduce how classification importance of an instruction is computed. In Section 3.2, we illustrate the challenges and our solutions when analyzing the semantics importance of an instruction.

3.1 Classification Importance Analysis

In the first step, we determine a set of instructions that are important to model classification results. As mentioned before, this is achieved by sampling N functions from the training dataset and studying function embedding changes caused by removing individual instructions. There are other alternatives to infer input importance in the literature of Deep Learning, e.g., by analyzing gradients [4, 47, 53] and attentions [11, 31, 59]. However, Transformer models have a discrete tokenization step which makes back-propagating gradients to the input space challenging. In addition, there are usually many attention heads, each yielding different importance results. We hence consider the model as a black box and observe how embeddings are changed by instruction removal. Note that if after removing an instruction, function embeddings drastically change, similarity query results likely have significant changes as well. The removed instruction is hence important to the classification results.

Formally, given an instruction i in a function f, the classification importance of i regarding the function, noted as $\mathcal{I}_c$, is defined as follows.

$$\mathcal{I}_c = 1 - cos\big(\mathbf{emb}(f), \mathbf{emb}(f\backslash\{i\})\big) \tag{1}$$

In the above equation, cos denotes the cosine similarity between two embeddings. $\mathbf{emb}(f)$ and $\mathbf{emb}(f\backslash\{i\})$ denote the embedding for f and the embedding for the resulting function after removing i from f, respectively.

The overall classification importance is then derived from the N sampled functions leveraging the above per-function classification importance equation. The process is formally described by Algorithm. 1. The algorithm takes as input a model and N functions sampled from the training dataset, and produces a list of instructions of top-k classification importance regarding the whole dataset. It first selects the most important instructions in each function (lines 3–10). The variable `importantInstr` defined at line 2 is a counter. It counts how many times an instruction is selected as one of the most important instructions (regarding a function) in lines 8–10. This is achieved by an outlier analysis explained later. Then the counter is sorted, and the most frequent K instructions are returned (line 11). Empirically, we use $N = 200$ and $K = 4$. Our ablation study in Section 4.6 shows that the performance of DiEmph is consistent across different values of N and K.

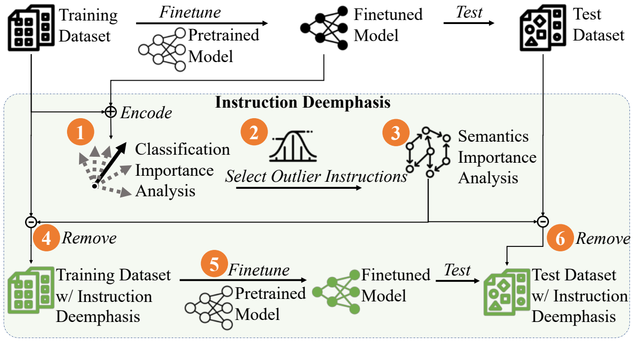

Training Dataset
Finetune
Pretrained Model
Finetuned Model
Test
Test Dataset
Encode
Instruction Deemphasis
1
Classification Importance Analysis
2
Select Outlier Instructions
3
Semantics Importance Analysis
4
Remove
6
Remove
Training Dataset w/ Instruction Deemphasis
5
Finetune
Pretrained Model
Finetuned Model
Test
Test Dataset w/ Instruction Deemphasis

Figure 3: Workflow of DiEmph. At the top is a typical training pipeline of Transformer model. Our technique is shown in the light green box. Major steps are marked in orange circles.

To select in a function the instructions with an exceptionally large classification importance, the algorithm uses the Kernel Density Estimation (KDE) (line 8) that finds outliers from a distribution. KDE fits a continuous probabilistic distribution given a list of discrete data points. Then it detects outliers by computing the probabilities that the corresponding importance values appear. Specifically, given an instruction i, it computes $\mathbf{P}(X < \mathcal{I}_c(i))$, where X is a random variable following the fitted distribution, and $\mathbf{P}$ denotes probability that X is less than the classification importance of i. If $\mathbf{P}(X < \mathcal{I}_c(i))$ is close to 1, it means that the classification importance of i is significantly larger than other instructions.

3.2 Semantics Importance Analysis

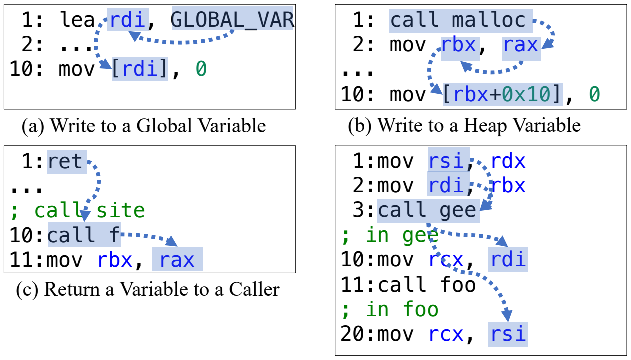
After deciding a set of instructions that have classification importance, the next step is to identify their semantics importance. This is achieved by identifying the stable variables in the N sampled functions, computing the backward program slices of these variables through a specially designed algorithm, and calculating the frequencies of instructions in the slices. That is, if an instruction frequently occurs in the backward slices of stable variables, it is considered to have semantics importance (regarding the whole dataset).

3.2.1 *Detection of Stable Variables.*

In our setting, stable variables include global, heap, return variables, and variables passed as function actual arguments. Stable variables are not sensitive to compiler and compilation option changes. For example, a heap variable tends

Algorithm 1: Selecting Instructions with High Classification Importance

```
Function select(model, functions)
    // importantInstr is a map from instructions to integers.
    // It counts how may times an instruction is considered
        important. Initially, all instructions are mapped to 0.
    importantInstr = {}
    for f ∈functions do
        allInstrs = []
        for $i \in f$ do
            // emb($f$) denotes using model to encode $f$
            $\mathcal{I}_c(i) = \cos(\text{emb}(f), \text{emb}(f \setminus \{i\}))$
            allInstrs.append($i$, $\mathcal{I}_c(i)$ )
        outliers = KDE(allInstrs)
        for $i$, _ ∈ outliers do
            importantInstr[$i$] += 1
    return largestK(importantInstr)
```



```
 1: lea rdi, GLOBAL_VAR
 2: ...
10: mov [rdi], 0
```

(a) Write to a Global Variable

```
 1: call malloc
 2: mov rbx, rax
...
10: mov [rbx+0x10], 0
```

(b) Write to a Heap Variable

```
 1:ret
...
; call site
10:call f
11:mov rbx, rax
```

(c) Return a Variable to a Caller

```
 1:mov rsi, rdx
 2:mov rdi, rbx
 3:call gee
; in gee
10:mov rcx, rdi
11:call foo
; in foo
20:mov rcx, rsi
```

(d) Pass a Variable to a Callee

Figure 4: Examples about how DiEmph identifies stable variables. Dashed blue lines indicate data flows in the code snippets.

to stay as a heap variable with compilation changes. Since we directly deal with binary executables, variable symbolic information is not available. We hence perform the following analysis to recognize the stable variables.

Identifying Global and Heap Variables. Global and heap variables are recognized by identifying whether a memory access instruction accesses a global or heap address. This is achieved by performing backward slicing on the address operand, typically a register, and checking if it is based on a global address, which is reflected by a constant address value in the global memory region, or on the return value of a heap allocation function. For example, in Fig. 4a, the register `rdi` at line 10 is used to compute a memory address. By tracing back the data flow, DiEmph finds that `rdi@10` is defined at line 1. The `lea` instruction at line 1 copies the address of a global variable (GLOBAL_VAR) to `rdi@1`[2], Similarly, DiEmph uses the same backward slicing technique to identify accesses to heap variables. Take Fig. 4b as an example. The memory address at line 10 is computed by `rbp+0x10`. `Rbx@10` is defined at line 2 by `rax@2`. `Rax@2` is the return value of the call (at line 1) to function `malloc`, which allocates a piece of heap memory. Thus `rbx@10` denotes a heap variable.

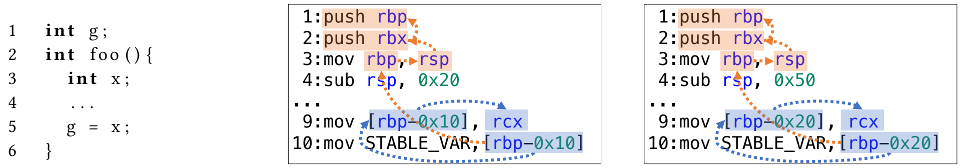
Identifying Function Return Variables. Return variables are considered stable variables since they denote function outputs. In x86 binary, an `ret` instruction does not have any operand. If there is a return value, it is by default stored in register `rax` (by instructions preceding the return instruction). The challenge lies in that a function that does not have an explicit return variable also uses `ret` to exit its execution. It is hence challenging to decide if a function has any return variable. Note that a simple method that searches for any write to `rax` before `ret` can hardly work as `rax` is a commonly used register in regular computation. Thus, when DiEmph analyzes an `ret` instruction in a function, it further analyzes invocations to this function (in other functions) and checks whether `rax` is being used right after the invocations. The intuition is that if the function has a return value, the value tends to be used after the function call (e.g., assigning to some variable). Particularly, for each invocation, if the register `rax` is used after the call without a new definition, DiEmph marks that the (invoked) function has a return value. For example, as shown in Fig. 4c, DiEmph tries to analyze whether the return instruction at line 1 in function f returns a value to the caller. Suppose it finds an invocation of f at line 10. After the call to f, at line 11, `rax` is used without redefinition. This indicates the variable denoted by `rax` is returned from the function f. Note that if DiEmph cannot find an invocation for the function, it conservatively assumes that the function has a return variable.

Identifying Function Actual Arguments. If a variable is being passed as an actual argument to some function, it is considered stable. However, without symbolic information, function signature is not available and variables passed as actual arguments are not explicit. We leverage the compiler convention that the first six actual arguments are passed from a caller to a callee using registers. This convention is true for all mainstream compilers as far as we know. As such, by checking data flow through registers across function call boundary allows us to identify variables that are passed as function arguments. For example, in Fig. 4d, the program calls to *gee* at line 3. DiEmph steps into the function *gee*, and finds that `rdi@10` is used before definition. That indicates `rdi` is a parameter of *gee*. As such, `rdi@2` is a stable variable. In some cases, the use of parameter-passing registers may not always be directly visible in the callee. For instance, in Fig. 4d, the register `rsi` is not used inside *gee*. However, function *gee* calls another function foo (line 11), and the parameters of foo are the same with *gee*. In this case, the compiler does not repeat the code for parameter passing. Instead, the variables in `rsi@10` and `rdi@10` are implicitly passed to foo. Note that at line 20, the register `rsi` is used before definition, indicating `rsi@20` is an argument. To handle such cases, DiEmph recursively traverses the call graph of a binary project and detects direct and transitive parameter passing.

3.2.2 *Semantics Importance via Binary Slicing.*

After identifying all stable variables, DiEmph measures the semantics importance of

[2] Here, GLOBAL_VAR is a token introduced by our preprocessor that considers any constant value in address loading instructions like `lea` as a global variable.



```
1   int g;
2   int foo(){
3     int x;
4     ...
5     g = x;
6   }
```

```
 1:push rbp
 2:push rbx
 3:mov rbp, rsp
 4:sub rsp, 0x20
...
 9:mov [rbp-0x10], rcx
10:mov STABLE_VAR, [rbp-0x10]
```

```
 1:push rbp
 2:push rbx
 3:mov rbp, rsp
 4:sub rsp, 0x50
...
 9:mov [rbp-0x20], rcx
10:mov STABLE_VAR, [rbp-0x20]
```

(a) Source Code of `foo()` **(b) Optimized Version** **(c) Un-optimized Version**

Figure 5: Example for stack variable accesses in binary programs. (a) shows a code snippet for function `foo()`**. (b) and (c) are two possible resulting binary programs after compilation. (b) is an optimized version with fewer variables on stack. The variable** `x` **is located at** `[rbp-0x10]`**. (c) is an un-optimized version with more variables on stack, and** `x` **is located at** `[rbp-0x20]`**.**

Algorithm 2: Computing Semantics Importance

```
1  Function getSemanticsImportance(function, stableVars)
       // instructionToCounter maps instructions to integers.
       // It counts how many stable variables an instruction may
           affect. Initially, all instructions are mapped to 0.
2      instructionToCounter = {}
3      for v ∈ stableVars do
4          backSlice = backwardSlicing(function, v)
5          for i ∈ function do
6              if i ∈ backSlice then
7                  instructionToCounter[i] += 1
8      semanticsImportance = {}
9      for i ∈ function do
10         semanticsImportance[i] = instructionToCounter[i] /
             len(importantVars)
11     return semanticsImportance
```

each instruction based on the backward slices of these stable variables (within the N sampled functions). The procedure is defined in Algorithm 2. The algorithm takes as input a binary function and a set of stable variables. It computes and returns the semantics importance for each instruction. It maintains a counter (line 2) for each instruction, and increases the counter by 1 every time the instruction is found in the slice of a stable variable (line 6). Finally, the importance for each instruction is computed as the ratio between the number of stable variables it may affect and the number of all stable variables.

A prominent challenge in slicing a binary program is to handle memory accesses. Different from source code, a binary program accesses most variables by their addresses. Therefore, statically determining if a memory read is dependent on a memory write entails precisely determining the set of addresses these accesses may refer to. This is a hard problem, especially at the binary level. Existing methods such as *value set analysis* [2] and *stochastic analysis* [72] are either imprecise or unsound. DiEmph resorts to a conservative solution and considers that any read to a global or a heap region is potentially dependent on any write to the same region, without disambiguating the addresses within the region. In contrast, for accesses to stack memory, which usually denote local variable or function argument accesses, we precisely determine their symbolic addresses and compute precise dependences. The rationale of having an over-approximate solution for global and heap accesses is that we are collecting the slices for all global and heap variables anyway.

Precluding Stack Address Dependences. A special feature of our slicing algorithm is that we preclude dependences induced by address computation if the address is on stack. Note that although local variables, usually allocated on stack, do not belong to stable variables, their accesses may be involved in the slices of stable variables, e.g., when a local loop variable i is used to index a global array $A[i]$. In fact, the slice of a stable variable usually includes a larger number of local variable accesses on stack. Although the inclusion of these accesses in the slice is completely correct, the compilation convention for such accesses may lead to substantial distribution biases. For example, while a local variable read is as simple as the presence of the variable in an expression, at the binary level, the read is broken down into multiple instructions such as computing the appropriate stack address and then performing the read. The stack address computation itself may include multiple instructions. Strictly following the standard slicing algorithm, these instructions should all be included in the slice and hence have semantics importance. However, they are in fact semantically unimportant as they are just stack access conventions. Depending on the stack memory layout optimizations, the compiler may place a variable in different stack locations and use different instruction patterns to compute the address before any access. Such differences shall be neutralized. Hence in DiEmph, we use data-flow analysis to determine if an address involved in a memory access denotes a stack address. If so, we preclude the dependence through the address operand from the slice. In the following, we use an example to illustrate the problem and then explain the data-flow analysis.

Example. Fig. 5 shows a function `foo()` in 5a and two versions of its compiled code in 5b and 5c. In 5a, the program declares a local integer variable x. It also has a global variable g, which is a stable variable according to our definition. The local variable is used in the computation of g (line 5) and hence included in the slice of g. Lines 1–4 in 5b and 5c show a typical function prologue. At first, the current values of the stack base register `rbp` and a general register `rbx` are stored on the stack (lines 1–2). In x86, there are two stack registers: the stack base pointer `rbp` pointing to the start of the previous stack frame (the stack frame of the caller function) and the stack pointer `rsp` denoting the end of the stack frame. Hence the region between `rbp` and `rsp` denotes the whole stack frame. Note that stack allocation is from high address to low address such that the value of `rsp` is smaller than that of `rbp`. A `push` instruction saves a value of a register to the memory location pointed to by the stack pointer `rsp` and automatically reduces `rsp` by 8 (assuming a 64bit machine). The saved values will be restored before returning from `foo()` such that the caller's stack frame can be re-activated. After saving, the registers can be safely updated inside `foo()`.

Take 5b as an example. Lines 3–4 set up a new stack frame for `foo()`. In particular, the base pointer is set to the end of the previous stack frame (line 3) and the end of the new stack frame is set to an address that is 32 smaller. As such, a 32-byte stack region delimited by the new `rbp` and `rsp` values is allocated. In lines 9 and 10, a stack address falling into the frame `rbp-0x10` is accessed in the computation of g, which is normalized to a symbol `STABLE_VAR` by our preprocessor. Observe that due to different stack layout strategies (for optimization purposes), the stack addresses of x are different in 5b and 5c, and the instruction encodings of the accesses are hence also different.

The arrows in 5b and 5c denote the program dependences. Blue arrows denote the dependences between two variables in the source code, and orange ones denotes the address dependences (induced by `rbp`). Note that there is dependence between instructions on line 2 and line 3 because line 2 implicitly subtracts `rsp`. If we included the orange dependences, lines 1–3 (those in the orange shade) would be included in the slice of g. In fact, all the stack accesses in `foo()` are transitively dependent on lines 1–3, leading to a conclusion that these were semantically very important instructions. However, they are just low-level artifacts that do not correspond to source-level semantics. Therefore, DiEmph precludes all dependences through stack address operands. Our ablation study in Section 4.6 demonstrates the importance of such strategy. □

The technical challenge lies in recognizing all operands denoting stack addresses. We achieve this by a data-flow analysis. A naive algorithm that simply finds all uses of `rbp` and `rsp` and rules out their dependences does not generalize well. For example, in the newer versions of GCC, the register `rbp` might be used as a normal register [20]. In this case, removing all dependencies with `rbp` may skip important dependencies. On the other hand, a compiler may copy stack pointers to other locations [63, 64].

We thus use Algorithm 3 to prune the stack address dependences. The algorithm performs a conservative data flow analysis to decide whether a variable contains a stack address of the stack. To simplify the discussion, we assume the binary program is lifted to an SSA form [51] so that we can directly deal with variables instead of registers or addresses. Intuitively, for a variable v at a program point p, the algorithm considers v contains a stack address if and only if v contains a stack address in *all* paths to p. The algorithm (line 14) takes a function as the input, and outputs variables that contain stack addresses at each instruction. Lines 15–19 initialize the data structures used in the analysis. The loop at line 20 iteratively propagates the analysis results, and terminates when the results converge. Line 15 defines two maps from instruction to set of variables: *stackAddrIn[i]* and *stackAddrOut[i]* record the set of variables containing a stack address at the program points before and after i, respectively. Initially, stackAddrIn maps all instructions to the empty set, and stackAddrOut maps all instructions to the universal set (to handle loops). At the entry point, only `rsp` stores an address to the stack.

For the program point before each instruction i, the algorithm intersects the analysis results from all the predecessors of i (line 2).

Algorithm 3: Pruning Stack Address Dependences

```
1  Function merge(i)
2      stackAddrIn[i] = $\bigcap_{p \in pred(i)}$ stackAddrOut[p]
3  Function propagate(i)
4      stackAddrOut[i] = stackAddrIn[i]
5      definedVar = def(i)
6      if $i$ defines an address then
7          for var ∈ uses(i) do
8              if var ∈ stackAddrIn[i] then
9                  stackAddrOut[i].add(definedVar)
10                 return
11     if definedVar ∈ stackAddrIn[i] then
12         stackAddrOut[i].remove(definedVar)
13     return
14 Function analyze(f)
15     stackAddrIn, stackAddrOut = {}, {}
16     for $i \in f$ do
17         stackAddrIn[i] = $\emptyset$
18         stackAddrOut[i] = $\top$
19     stackAddrIn[f.entry] = {rsp}
20     while True do
21         prevStackAddrOut = stackAddrOut
22         for $i \in f$ do
23             merge(i)
24             propagate(i)
25         if prevStackAddrOut == stackAddrOut then
26             break
27     return stackAddrIn
```

Our analysis assumes only arithmetic instructions and copy instructions between variables can be used to calculate a stack address. For an instruction that may define a stack address (line 2), if at least one of the used variables contains a stack address, the analysis considers this instruction defines a variable containing stack address (line 9). Otherwise, the variable defined by this instruction kills the previous definitions that are in the set stackAddrOut (line 11).

4 EVALUATION

DiEmph is implemented on IDA Pro [23] and PyTorch [48]. We evaluate our technique via the following research questions (RQs):

RQ1: Can DiEmph help binary similarity models achieve better performance when the compiler configurations of the test dataset are different from the training dataset? We say that the test data are *out-of-distribution*. It denotes a realistic and challenging use scenario for binary analysis tools.

RQ2: Is DiEmph effective with different pool sizes of the candidate functions?

RQ3: How does DiEmph affect performance of models when the compiler configurations of the test dataset align with those of the training dataset? In this case, we say the test data are *in-distribution*.

RQ4: How much time does DiEmph take to analyze functions in training datasets?

RQ5: How does each component of DiEmph affect the performance?

We also conduct a case study to demonstrate the effectiveness of DiEmph on out-of-distribution test data.

4.1 Experiment Setup

We conduct the experiments on a server with a 24-core Intel Xeon 4214R CPU at 2.40 GHz, 188 G memory, 8 Nvidia RTX A6000 GPUs, and Ubuntu 18.04.

Baseline Models. We train two state-of-the-art Transformer similarity analysis models, i.e., JTrans [61] and Trex [44] on three well-known public datasets, resulting 6 baseline models in total. Although there are other binary code similarity methods and some are based on Deep Learning models as well, it was shown that the Transformer based methods (i.e., our baselines) outperform these techniques [44, 61]. For each model, we use the model pretrained by its authors, and finetune the model on three different well-known public datasets for the binary similarity task. The first dataset is BinaryCorp-3M [61]. It contains around 3 million binary functions (which correspond to around 600 k functions in source code). The dataset is compiled by GCC with 5 different optimization flags. The second dataset is BinKit [27]. It contains 51 GNU projects compiled with 9 different compilers × 4 different optimization flags. It has around 4.5 million binary functions (which correspond to around 126 k source code functions). The third dataset is HowSolve [35]. It contains 7 projects compiled with 8 compilers × 5 optimization flags, which correspond to around 4.4 million binary functions (around 110 k source code functions). Note that for the later two datasets, we only use the binary programs compiled for the x64 architecture. For each model/dataset setting, we use the same set of hyper-parameters to finetune both the original and the DiEmph-enhanced models. For most hyper-parameters (e.g., learning rate), we use the default values coming with the models. We tune the number of epochs due to the significant difference in dataset sizes. Our training scripts are publicly available at [65], and the key hyper-parameters are listed in Section A of the supplementary material[66].

Test Datasets. We build two test datasets from the real-world projects commonly-used in binary similarity analysis [15, 21, 27, 35, 44]. We first recognize projects that are used by at least two of existing work [15, 21, 27, 35, 44], and then we filter out the projects that contain less than 500 binary functions. Our datasets hence consist of 7 real-world projects. They are Curl, Coreutils, Binutils, ImageMagick, SQLite, OpenSSL and Putty. Dataset-I: Binary programs in the first dataset are compiled by GCC-7.5 with -O0 and -O3 optimization flags. They are considered as test data within the distribution of the training dataset because (1) all training datasets contain binary programs compiled with GCC (2) all training datasets contain binary programs compiled by a compiler newer than GCC-7.5. That indicates all the optimizations/conventions by GCC-7.5 are very likely present in the training dataset. We obtain the binaries from [44]. Dataset-II: Binary programs in the second dataset are compiled by GCC-9.4 with -O3 flag and Clang-10 with -O0 flag. They are considered as test data with compiler configurations different to the training datasets because programs in the training datasets are compiled with older versions of compilers. The two new compilers introduce new optimizations and likely emit instructions with new patterns [20, 32]. For simplicity, we refer to Dataset-I as the *In-Distribution* dataset, and refer to Dataset-II as the *Out-of-Distribution* dataset. We want to point out that we exclude all projects in test datasets from the training datasets and make sure there are no overlapping functions between the test datasets and the training datasets.

Metrics. In this section, we use precision at 1 (PR@1) as our metrics. Suppose that we iteratively query a set of binary functions from a pool of candidate functions. PR@1 measures in how many queries, the correct function (i.e., the function compiled from the same source code as the query function) is returned as the most similar function. Note that our experiments are conducted in a way that each function has only one similar function (more details later in the section). We also evaluate DiEmph in terms of PR@5, PR@10, and Mean Reciprocal Rank (MRR). The results are shown in Section B of the supplementary material[66] for brevity.

4.2 RQ1: Performance Improvement on the Out-of-Distribution Dataset

In this section, we evaluate whether DiEmph can help models achieve better performance on the out-of-distribution dataset. For each baseline model, DiEmph takes as input the model and the training dataset, and outputs a list of instructions need to be deemphasized. To obtain an improved model, we remove the top-four most frequently occurring problematic instructions from the training dataset and rerun the finetune process. We then test the performance for both the baseline model and the improved model on the out-of-distribution dataset. For each binary program in our dataset, we randomly sample 500 functions from the O0 binary, and query them one by one in a pool consisting of the corresponding 500 functions from the O3 binary. In other words, there is only one similar function in the pool for each query. Such a setup is consistent with the literature [22, 35, 62]. During testing, we also remove the deemphasized instructions from the test inputs to the improved model for input space consistency.

The results are shown in Table 1. We can see that DiEmph improves the PR@1 of models by 3.7-14.4%. Note that these improvements are considered significant for the binary similarity task due to its challenging nature. In the binary similarity literature, the improvement to the baseline method is usually 3-8% [15, 62]. We can observe that the improvement is most prominent on the JTrans model trained with the BinaryCorp-3M dataset. DiEmph improves it by 14.4%. The improved JTrans model trained with the deemphasized BinaryCorp-3M denotes the new state-of-the-art. The fact that we are able to achieve substantial improvement on the most complex model and the largest dataset indicates the value of instruction deemphasis in practice. The improvements are around 6% and 4% for the models trained with the How-Solve dataset and the BinKit dataset, respectively. The improvements on models trained with the BinKit dataset are lower because BinKit contains only around 126 k source code functions. Although each function is compiled with many different configurations, the limited program diversity leads to limited generalizability. The improvement on the Trex model trained with the BinaryCorp-3M dataset is relatively low (3.7%). That is because the BinaryCorp-3M dataset contains many functions with relatively complex control-flow structures, and Trex does not precisely encode control flow information such that it may not learn control flow features well.

Table 1: Performance (PR@1) improvement on the out-of-distribution dataset. The first column lists the name of binary programs. The model setups are listed in the first row. Each model is denoted with its architecture and the training dataset, in the form of $ModelArch_{dataset}$. BC, BK, and HS denote BinaryCorp-3M, Binkit, and How-solve, respectively. In the second row, for each model, Ori. means the PR@1 for the original model, DiEmph means the PR@1 for the model improved by DiEmph, and Impr. means the improvement achieved by applying DiEmph to the model.

Programs	$JTrans_{BC}$			$JTrans_{BK}$			$JTrans_{HS}$			$Trex_{BC}$			$Trex_{BK}$			$Trex_{HS}$		
	Ori.	DiEmph	Impr.	Ori.	DiEmph	Impr.	Ori.	DiEmph	Impr.	Ori.	DiEmph	Impr.	Ori.	DiEmph	Impr.	Ori.	DiEmph	Impr.
Curl	42.6	61.8	19.2	54.0	55.6	1.6	50.6	50.6	0.0	48.8	50.2	1.4	35.6	40.2	4.6	25.8	33.8	8.0
Binutils	37.6	53.2	15.6	49.8	54.4	4.6	36.4	40.4	4.0	37.6	44.0	6.4	33.8	34.8	1.0	21.4	27.0	5.6
Coreutils	31.4	42.6	11.2	37.4	41.8	4.4	32.6	34.0	1.4	33.4	40.2	6.8	28.0	31.6	3.6	22.4	28.2	5.8
ImageMagick	22.6	39.0	16.4	42.2	46.2	4.0	30.6	43.4	12.8	39.6	45.8	6.2	27.6	33.8	6.2	22.8	29.6	6.8
SQLite	42.8	60.0	17.2	51.4	56.8	5.4	42.4	56.0	13.6	65.6	66.8	1.2	44.0	48.2	4.2	32.6	40.2	7.6
OpenSSL	47.6	53.8	6.2	46.2	52.0	5.8	54.8	61.0	6.2	46.2	50.2	4.0	32.8	38.4	5.6	28.4	33.0	4.6
Putty	35.8	50.6	14.8	36.8	39.2	2.4	42.0	45.4	3.4	39.2	38.6	-0.6	34.4	37.2	2.8	27.0	33.8	6.8
Average	**37.2**	**51.6**	**14.4**	**45.4**	**49.4**	**4.0**	**41.3**	**47.3**	**6.0**	**44.3**	**48.0**	**3.7**	**33.7**	**37.7**	**4.0**	**25.7**	**32.2**	**6.5**

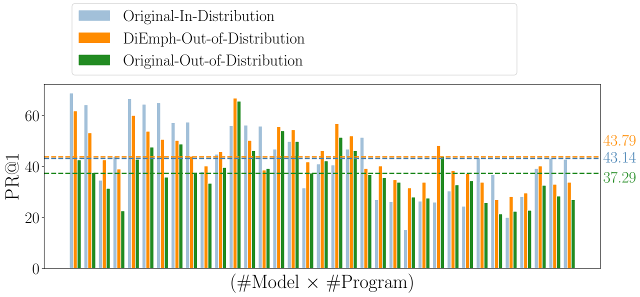


Figure 6: DiEmph helps models alleviate performance degradation. Each bar in the figure shows the performance of one model on one test program. The y axis denotes PR@1. The dashed lines show the average performance.

To normalize the effectiveness of DiEmph w.r.t. the differences introduced by model architectures and training datasets, we additionally run the baseline models on the in-distribution dataset. This allows us to assess the extent to which DiEmph can mitigate the performance degradation caused by the distribution shift between the test and training data (specifically, different compiler configurations). The results are visualized in Fig. 6. On average, we can see that the performance of the original models degrades by 5.9% due to distribution shift. With the enhancement provided DiEmph, models' performance becomes comparable to the original models' performance on the in-distribution dataset. That indicates that DiEmph facilitates better generalization of these models, mitigating the performance degradation caused by the distribution shift.

Note that the performance of the models discussed in this section refers to the checkpoints that achieved the *highest* performance during the training processes. Additionally, we further analyze the performance of each model at each checkpoint and observe that DiEmph consistently enhances the generalizability of the model across all checkpoints. Details are shown in Fig. 10 of the supplementary material[66].

4.3 RQ2: Effectiveness with Different Pool Sizes

The performance of a binary similarity model may vary when the size of candidate function pool is different [61]. Thus we validate whether DiEmph can effectively improve model performance (on the out-of-distribution dataset) with different pool sizes. For each model, we test its performance on 7 function pools with different sizes from 16 to 500. We show in Fig. 7 the test results for the JTrans model trained on BinaryCorp-3M. The results for other models are shown in Fig. 11 of the supplementary material[66]. We can see that DiEmph can effectively improve the original model's performance on all different pool sizes. The improvement is more significant when the pool size is larger than 100. That is because the task of finding similar functions becomes harder when there are more candidate functions. It requires the model to more precisely distinguish functions based on instructions with important semantics. Note that we run each test for 10 times, and Fig. 7 shows the average results.

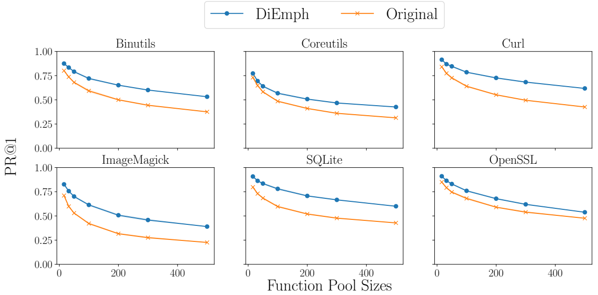


Figure 7: Effectiveness of DiEmph (for $JTrans_{BC}$) with different pool sizes. Each figure shows the performance of two models on a program from the out-of-distribution dataset. The x-axis denotes the sizes of candidate function pool. The y-axis denotes PR@1.

4.4 RQ3: Effects on In-Distribution Data

In this section, we study how DiEmph affects the performance on the in-distribution dataset. Following the setup of RQ1, we run both the original model and the model improved by DiEmph on the in-distribution dataset. The results show that DiEmph is also able to slightly improve models' performance (by around 3% on average) on the in-distribution dataset. That is because DiEmph removes from the training datasets instructions with high classification importance but low semantics importance. This helps the models

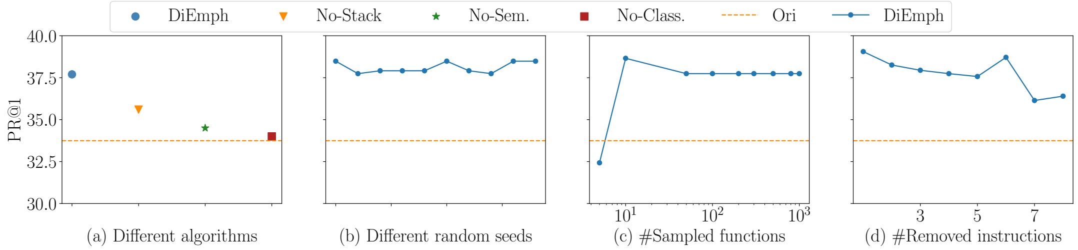


Figure 8: How the performance of DiEmph changes across different configurations

learn to represent a program based on the semantically important features. Compared with the improvement DiEmph achieves on the out-of-distribution data, the improvement on in-distribution data is smaller. That is because the distribution of the test dataset is similar to the training dataset. A baseline model may already achieve relatively good performance even if it represents programs based on semantically unimportant instructions. Details are shown in Section C of the supplementary material[66].

4.5 RQ4: Run Time Efficiency

We use DiEmph to analyze the six models and test whether it is time efficient. Specifically, for each model, DiEmph randomly samples 200 functions from the training data, computes the classification importance for each instruction, uses KDE to select outliers, and validates whether instructions with high classification importance are semantically important. The results show that it takes 29 minutes for DiEmph to analyze one model. Time time consumption of DiEmph is acceptable since it is a one-time effort. Details are in Section D of the supplementary material[66].

4.6 RQ5: Ablation Study

To analyze the impact of each component on the performance of DiEmph, we conduct four ablation studies. These studies include varying the algorithm of DiEmph by disabling each component individually, testing DiEmph with different random seeds, altering the number of sampled functions, and altering the number of removed instructions. For each study, we run DiEmph (with different configurations) on the Trex model trained with the BinKit dataset, resulting a set of improved models. We then test these models on the out-of-distribution dataset. The results are shown in Fig. 8. The dashed yellow line in Fig. 8 represents for the performance of the original model.

Effects of each component. We construct three variants of DiEmph by disabling each component individually. Their performance is shown in Fig. 8a. *No-Stack* denotes the variant that does not trace the stack pointers and does not prune the dependencies introduced by stack operations. *No-Sem.* and *No-Class.* denotes the variants that do not analyze the semantics importance and do not analyze the classification importance, respectively.

We can see that the *No-Stack* variant is still able to improve the original model's performance, while the improvements are less significant than DiEmph. That is because without pruning the stack dependencies, the variant is more conservative when computing the semantics importance of instructions. That is, fewer semantically unimportant (but classification-wise important) instructions are removed from the dataset. Although the removal helps the model learn to encode programs based on semantically important instructions, the resulting model may still learn some undesirable bias in the training data. Thus it has worse generalization when tested on the out-of-distribution data.

The *No-Sem.* variant slightly improves the baseline model's performance as well. Although the *No-Sem.* variant does not analyze semantic importance of instructions, we observe that the set of removed instructions, selected based on high classification importance, includes instructions with low semantics importance. Removing these instructions helps the model generate better embeddings based on semantically important instructions. However, since the *No-Sem.* variant is unaware of the semantics importance of instructions, it also removes instructions that carry important semantics. This introduces noise to the training process, resulting in a less significant improvement compared to DiEmph.

The *No-Class.* variant shows almost no improvement over the original model. That is because the removed instructions, which generally have low classification importance, do not significantly affect the embeddings generated by the original model. Removing these instructions thus will not significantly impact the behavior of the original model.

Effects of random seeds. We run the sampling process in DiEmph with 10 different random seeds. The results are shown in Fig. 8b. We can see that DiEmph can consistently improve the baseline model across different random seeds.

Effects of the number of sampled functions. We change the number of functions sampled by DiEmph from 5 to 1000. The results are shown in Fig. 8c. We can see that the effectiveness of DiEmph is stable when the sample size is larger than 50. Thus the sample size of 200 used in our system can be considered as sufficient.

Effects of the number of removed instructions. Note that after finding the problematic instructions, DiEmph removes the top-K most frequently occurring problematic instructions. In this study, we alter K from 1 to 8. The results are shown in Fig. 8d. We can see that DiEmph is most effective when K is no larger than 6. That is because removing too many instructions may result in extremely short functions that do not contain enough semantics for the model to encode. We thus use $K = 4$ in our system, which effectively improves models' performance and meanwhile does not shorten functions too much.

4.7 Case Study

We conduct a case study with the JTrans model trained on the BinaryCorp-3M dataset to demonstrate how DiEmph improves the performance of the model on the out-of-distribution dataset.

By analyzing the model and the training dataset, DiEmph finds that an instruction "`or [rsp], 0`" has exceptionally high classification importance but low semantics importance in many functions. This instruction performs a bitwise-or operation between the variable stored in `[rsp]` and 0, and hence it does not modify the value of the variable. The instruction is inserted by an optimizing compiler to improve cache performance in modern processors.

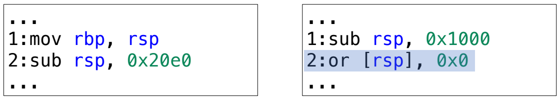
We show in Fig. 9 an example in the test dataset that contains this instruction. The instruction is highlighted with the blue shade in Fig. 9b. In Fig. 9b, note that the variable stored in `rsp` at line 2 is not initialized (because the value of `rsp` is updated at line 1). And the inserted instruction does not modify the variable either. Thus the following instructions in the function are not likely to use the definition `[rsp]@2`. That is, the instruction is not semantically important in this function. Also note that this instruction is not in the function compiled (from the same source code) with Clang -O0, shown in Fig. 9a.

The original JTrans model generates a similarity score as low as 0.45 for this pair of functions due to the difference of the highlighted instruction. After we remove it from the optimized function (Fig. 9b), the similarity score increases to 0.52. On the other hand, the JTrans model improved by DiEmph generates a similarity score as high as 0.69 for this pair of functions. That indicates DiEmph indeed helps the model better represent a program based on semantically important instructions.

5 RELATED WORK

Binary Similarity Analysis. Binary similarity analysis has critical security applications. Thus the community has made significant efforts in this area [17, 45, 62], leveraging both the dynamic features [17, 22, 62] and the static features [45] of programs. Moreover, recent advances in Deep Learning techniques have led to unprecedented capabilities in many areas, including binary similarity [14, 15, 27, 30, 35, 37, 38, 67]. Among them, two Transformer-based methods [44, 61] have been proposed, demonstrating superior performance compared to previous work, leveraging the recent success of Transformers [8, 49, 50, 58] in the NLP domain. These techniques represent notable advances in binary similarity analysis and have the potential to aid in the identification and mitigation of security threats. Our work is an enhancement to these approaches, aiming to improve the generalizability of Transformer-based models.

Model Debugging. Although deep learning models achieve significant success in various domains, their opaque nature poses challenges in understanding the root cause of unexpected behaviors (e.g., low accuracy). As a result, many researchers focus on the area of *model debugging*, which treat deep learning models as traditional software and develop approaches to analyzing and debugging deep learning models [34, 41, 57]. However, these methods do not address the challenges in improving Transformer models for binary code analysis due to the complexity of the Transformer models and the domain-specific constraints of binary programs. Our approach focuses on such complex binary code models, leveraging domain knowledge about binary similarity analysis and binary program slicing. Our method can pinpoint the code patterns introducing biases to models and help models focus on semantically important instructions.



```
...
1:mov rbp, rsp
2:sub rsp, 0x20e0
...
```

(a) Instrs. Compiled with Clang O0

```
...
1:sub rsp, 0x1000
2:or [rsp], 0x0
...
```

(b) Instrs. Compiled with GCC O3

Figure 9: Instructions in `head_lines()` of Coreutils. The two snippets of instructions are generated by Clang -O0 and GCC -O3, respectively. The instruction highlighted in the blue box is inserted by GCC. It is used to optimize the cache performance in modern processors.

6 THREATS TO VALIDITY

Our prototype DiEmph focuses on Transformer-based models and hence the reported results may not hold on other types of models such as GNN-based models. However, we believe our idea of semantic-driven instruction deemphasis has the potential to generalize to other model architectures. We leave the exploration as our future work.

DiEmph relies on disassembling tools (IDA-Pro) to analyze binary programs. The quality of disassembled code may affect DiEmph's performance, though SOTA disassembler achieves more than 95% accuracy[39, 42, 43, 69] in most cases. The reported results are achieved with the selected hyper-parameters, e.g., random selection of 200 functions. We have conducted a substantial ablation study to validate the stability of our results.

7 CONCLUSION

We develop a novel technique to improve the performance of Transformer based binary code similarity analysis models. The technique detects instructions that have undesirably biased distributions in the training dataset because of compiler conventions. It features a (Deep Learning) model classification importance analysis that determines if an instruction is important for model output and a program analysis based semantics importance analysis that determines if an instruction denotes part of essential program semantics. Instructions that have classification importance but not semantics importance are removed from the dataset. Finetuning the model on the updated dataset yields substantially better performance than the original models.

8 DATA AVAILABILITY

Our experimental data and the source code are available at [65].

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] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. 2019. A Systematic Review on Code Clone Detection. *IEEE Access* 7 (2019), 86121–86144. https://doi.org/10.1109/ACCESS.2019.2918202

[2] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.

[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] Oren Barkan, Edan Hauon, Avi Caciularu, Ori Katz, Itzik Malkiel, Omri Armstrong, and Noam Koenigstein. 2021. Grad-sam: Explaining transformers via gradient self-attention maps. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*. 2882–2887.

[5] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. https://doi.org/10.1109/ICSM.1998.738528

[6] 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.

[7] BinDiff 2022. *zynamics BinDiff*. https://www.zynamics.com/bindiff.html

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] 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.

[10] 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

[11] Hila Chefer, Shir Gur, and Lior Wolf. 2021. Transformer interpretability beyond attention visualization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 782–791.

[12] Coreutils 2022. *Coreutils - GNU core utilities*. https://www.gnu.org/software/coreutils/

[13] 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

[14] 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

[15] 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

[16] S. Ducasse, M. Rieger, and S. Demeyer. 1999. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*. 109–118. https://doi.org/10.1109/ICSM.1999.792593

[17] 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.

[18] 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.

[19] 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

[20] GNU. 2022. *gcc-9*. Retrieved Feb 16, 2023 from https://gcc.gnu.org/gcc-9/

[21] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. 2021. A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. *IEEE Transactions on Software Engineering (TSE)* 47, 6 (June 2021), 1241–1258. https://doi.org/10.1109/TSE.2019.2918326

[22] 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

[23] IDA Pro 2022. *A powerful disassembler and a versatile debugger*. https://hex-rays.com/ida-pro/

[24] 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)*.

[25] 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

[26] 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.

[27] 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

[28] Raghavan Komondoor and Susan Horwitz. 2001. Using Slicing to Identify Duplication in Source Code. In *Static Analysis*, Patrick Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 40–56.

[29] J. Krinke. 2001. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*. 301–309. https://doi.org/10.1109/WCRE.2001.957835

[30] 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

[31] Shengzhong Liu, Franck Le, Supriyo Chakraborty, and Tarek Abdelzaher. 2021. On exploring attention-based explanation for transformer models in text classification. In *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, 1193–1203.

[32] LLVM. 2020. *clang-10*. Retrieved Feb 16, 2023 from https://releases.llvm.org/10.0.0/

[33] 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

[34] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 175–186. https://doi.org/10.1145/3236024.3236082

[35] 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.

[36] 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

[37] 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

[38] Tomás Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

[39] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[40] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 253–270. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ming

[41] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4901–4911. https://proceedings.mlr.press/v97/odena19a.html

[42] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *SP*. IEEE, 833–851.

[43] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. In *NDSS*. The

Internet Society.

[44] 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

[45] 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

[46] 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

[47] Nina Poerner, Hinrich Schütze, and Benjamin Roth. 2018. Evaluating neural network explanation methods using hybrid documents and morphosyntactic agreement. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 340–350.

[48] PyTorch 2023. *An open source machine learning framework that accelerates the path from research prototyping to production deployment.* https://pytorch.org

[49] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training. (2018).

[50] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[51] rev.ng. 2023. *Rethink Binary Analysis.* Retrieved Feb 16, 2023 from https://rev.ng

[52] 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

[53] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2017. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*. 618–626.

[54] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

[55] 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

[56] 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.

[57] Guanhong Tao, Shiqing Ma, Yingqi Liu, Qiuling Xu, and Xiangyu Zhang. 2020. TRADER: trace divergence analysis and embedding regulation for debugging recurrent neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 986–998. https://doi.org/10.1145/3377811.3380423

[58] 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

[59] Jesse Vig. 2019. A Multiscale Visualization of Attention in the Transformer Model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. 37–42.

[60] 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

[61] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. JTrans: Jump-Aware Transformer for Binary Code Similarity Detection. 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, 1–13. https://doi.org/10.1145/3533767.3534367

[62] 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.

[63] Yuting Wang, Pierre Wilke, and Zhong Shao. 2019. An Abstract Stack Based Approach to Verified Compositional Compilation to Machine Code. *Proc. ACM Program. Lang.* 3, POPL, Article 62 (jan 2019), 30 pages. https://doi.org/10.1145/3290375

[64] Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: Verified Separate Compilation of C Programs into ELF Object Files. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 197 (nov 2020), 28 pages. https://doi.org/10.1145/3428265

[65] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. *Artifact for DiEmph.* https://doi.org/10.5281/zenodo.7978808

[66] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. *Supplementary Material.* Retrieved May 27, 2023 from https://github.com/XZ-X/DiEmph/blob/master/suppl-material.pdf

[67] Xi Xu, Qinghua Zheng, Zheng Yan, Ming Fan, Ang Jia, and Ting Liu. 2021. Interpretation-Enabled Software Reuse Detection Based on a Multi-level Birthmark Model. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 873–884. https://doi.org/10.1109/ICSE43902.2021.00084

[68] 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

[69] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. 2022. D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 728–745.

[70] 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

[71] 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

[72] 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, Article 137 (oct 2019), 31 pages. https://doi.org/10.1145/3360563

Received 2023-02-16; accepted 2023-05-03