IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 22 February 2023, accepted 13 March 2023, date of publication 20 March 2023, date of current version 28 March 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3259481

RESEARCH ARTICLE

Binary Code Representation With Well-Balanced Instruction Normalization

HYUNGJOON KOO[1], SOYEON PARK[2], DAEJIN CHOI[3], AND TAESOO KIM[2]
[1]Department of Computer Science and Engineering, Sungkyunkwan University, Suwon 16419, South Korea
[2]School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA
[3]Department of Computer Science and Engineering, Incheon National University, Incheon 22012, South Korea

Corresponding author: Hyungjoon Koo (kevin.koo@skku.edu)

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) Grant funded by the Ministry of Education of the Government of South Korea under Grant 2022R1F1A1074373; in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grants funded by the Korean Government (The Ministry of Science and Information Communication Technology (MSIT); Graduate School of Convergence Security, Sungkyunkwan University) under Grant 2022-0-01199; in part by the National Science Foundation (NSF) under Award CNS-1749711; in part by the Office of Naval Research (ONR) under Grant N00014-23-1-2095; in part by the Defense Advanced Research Projects Agency (DARPA) Verified Security and Performance Enhancement of Large Legacy Software (V-SPELLS) under Grant N66001-21-C-4024; and in part by the gifts from Facebook, Mozilla, Intel, VMware, and Google.

ABSTRACT The recovery of contextual meanings on a machine code is required by a wide range of binary analysis applications, such as bug discovery, malware analysis, and code clone detection. To accomplish this, advancements on binary code analysis borrow the techniques from natural language processing to automatically infer the underlying semantics of a binary, rather than replying on manual analysis. One of crucial pipelines in this process is instruction normalization, which helps to reduce the number of tokens and to avoid an out-of-vocabulary (OOV) problem. However, existing approaches often substitutes the operand(s) of an instruction with a common token (*e.g.*, callee target → FOO), inevitably resulting in the loss of important information. In this paper, we introduce *well-balanced instruction normalization* (WIN), a novel approach that retains rich code information while minimizing the downsides of code normalization. With large swaths of binary code, our finding shows that the instruction distribution follows Zipf's Law like a natural language, a function conveys contextually meaningful information, and the same instruction at different positions may require diverse code representations. To show the effectiveness of WIN, we present DeepSemantic that harnesses the BERT architecture with two training phases: pre-training for generic assembly code representation, and fine-tuning for building a model tailored to a specialized task. We define a downstream task of binary code similarity detection, which requires underlying code semantics. Our experimental results show that our binary similarity model with WIN outperforms two state-of-the-art binary similarity tools, DeepBinDiff and SAFE, with an average improvement of 49.8% and 15.8%, respectively.

INDEX TERMS Binary code, code representation, BERT, well-balanced instruction normalization, binary code similarity detection.

I. INTRODUCTION

In today's computing environment, encountering binary-only software is common including commodity or proprietary programs, system software like firmware and device drivers. Accordingly, binary analysis is essential in implementing a wide range of popular use cases [11], [13], [21], [65]: *e.g.*, detecting code clone or software plagiarism to protect against intellectual property infringement [17], [40], [67], discovering vulnerabilities in distributed software [7], [8], [14], [15], [19], [38], [42], [51], [52], [57], [58], detecting [5], [6], [35] and classifying [28], [34] malware, and analyzing program repairs or patches [20], [29], [64], and establishing toolchain provenance [48], [56] for digital forensics purposes.

However, analyzing a binary code is inherently more challenging than analyzing source code, as it requires inferring contextual meanings from machine-interpretable code alone. Unlike human-readable source code, binary code is a final

The associate editor coordinating the review of this manuscript and approving it for publication was Seok-Bum Ko.

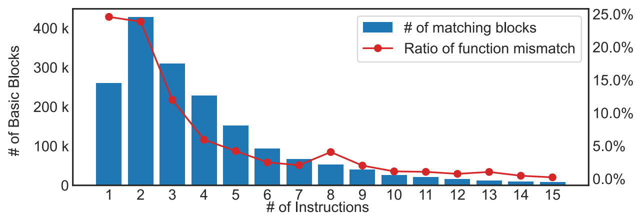


FIGURE 1. Histogram for the number of matching block pairs (left label) and the ratio of function mismatch (right label) by the number of instructions per basic block with DeepBinDiff [67] from our evaluation dataset (section VI). The function mismatch represents basic block pairs with a sequence of identical instructions that do not belong to the same function. More than eight out of 10 from the whole matching basic blocks incorporate 5 instructions or less.

product of a complex compilation process that involves massive transformations (*e.g.*, optimizations), including control flow graph alteration, function inlining, instruction replacement, and dead code elimination. As a result, much of high-level semantic information useful for analysis is disappeared. Besides, binary code generation is impacted by other major factors such as an architecture, compiler, compiler version, compiler option, and code obfuscation.

Recently, machine learning-based techniques [7], [11], [13], [17], [21], [36], [38], [43], [65], [67], [68], [69], [74] have emerged as a promising solution for addressing the challenge of recognizing code semantics in binaries. While traditional approaches, such as static analysis (*e.g.*, graph isomorphism on call graph [14], [19]) or dynamic analysis (*e.g.*, taint analysis [18], [51]), have demonstrated high accuracy in specific tasks, machine learning-based approaches offer significant advantages in rapidly changing computing environments. With a sufficient amount of training data, a single model can be leveraged for multiple platforms and architectures, and be continuously improved with an increasing number of new inputs. Indeed, recent state-of-the-art tools [17], [43], [67], [69], [74] have successfully generated code embeddings (vectors) for semantic clone detection, even across different architectures [69], [74], optimizations [17], [43], [67], and obfuscation techniques [17].

However, we raise a question about the practicality and effectiveness of code embedding to infer code semantics. Figure 1 illustrates the matching basic block pairs and function mismatch cases from our DeepBinDiff [67] evaluation dataset (Section VI). Our experiment reveals that 83% of matching block pairs contain five instructions or less, indicating that binary code embeddings may not convey sufficient meaning for the inference of a large block. Besides, we examine function mismatch cases because matching block pairs that consist of identical instructions can belong to a different function body. Figure 1 (red dots) shows that by and large, the ratio of function mismatch cases increases as a block size is smaller: This implies that using basic blocks as a granularity for a binary similarity task may be insufficient to deduce code semantics, especially for block pairs containing only `nop`, `jmp`, or `call` instructions.

A proper binary code representation is of significance because it is directly fed into learning a model as raw data. To further investigate, we analyze large swaths of binary code including approximately 108 million machine instructions or 1.7 million binary functions. First, our findings indicate that the instruction distribution follows Zipf's law [73], analogous to a natural language. Second, a function often conveys contextually meaningful information. Third, word2vec [44] lacks diverse representations for the identical instruction in different positions.

We introduce DeepSemantic, a system that leverages the BERT (Bi-directional Encoder Representations from Transformers) architecture [16] to infer code semantics. Based on our insights, DeepSemantic has been carefully designed to achieve our goals in four ways: i) *function-level granularity*; *e.g.*, the unit of an embedding is a binary function, ii) *function embedding as a whole*; *e.g.*, each instruction may have multiple representations depending on the location of a function, iii) *well-balanced instruction normalization (WIN)* that strikes a balance between too-coarse-grained and too-fine-grained normalization, and iv) a *two-phase training model* to support a wide range of other downstream tasks based on a pre-trained model. DeepSemantic mainly consists of two separate training stages. The first stage creates a one-time *generic code representation* (*i.e.*, pre-trained model or DS-Pre) that is applicable to *any* downstream task requiring code semantics inference at a pre-training stage. The second stage generates a *special-purpose code representation* (*i.e.*, fine-tuned model or DS-Task) for a given specific task based on the pre-trained model for fine-tuning. The first phase employs a general dataset with unsupervised learning, while the second phase uses a task-oriented dataset with supervised learning. The two-stage model in DeepSemantic allows for *re-purposing* a pre-trained model to quickly apply other downstream tasks using less expensive computational resources.

To show the effectiveness of DeepSemantic, we have applied it to a binary code similarity task (DS-BinSim). The empirical results show that DS-BinSim by far outperforms two state-of-the-art binary similarity comparison tools, obtaining a 49.8% higher F1 than DeepBinDiff [67] (up to 69%) and 15.8% than SAFE [43] (up to 28%) on average.

In summary, we make the following contributions:

- We thoroughly investigate binary code on a large scale to obtain fruitful insights for the inference of binary code semantics.
- We devise well-balanced instruction normalization (WIN) that can preserve as much contextual information as possible while maintaining efficient computation.
- We implement a simplified BERT architecture atop our observations and insights on binaries, which can generate semantic-aware code representations. Note that we will release our pre-trained model[1] to foster further research on binary code representation with deep learning.

[1]https://github.com/SecAI-Lab/win/

- We experimentally demonstrate both the effectiveness and efficiency of DeepSemantic with a binary similarity (DS-BinSim) task. In particular, DS-BinSim surpasses the two state-of-the-art binary similarity tools (*i.e.*, DeepBinDiff [67] and SAFE [43]).

II. BACKGROUND

This section describes how we integrate a handful of advanced concepts from natural language processing (NLP) literature to develop DeepSemantic.

A. BINARY CODE REPRESENTATION

Binary code is a compressed representation of machine instructions, expressed in 0s and 1s, after a complex compilation process. As the compilation process eliminates most high-level concepts such as variable names, structures, types, class hierarchies, deducing contextual meanings becomes extremely challenging.

B. RECURRENT NEURAL NETWORK

A recurrent neural network (RNN) is a specialized type of neural network designed to process sequential data (*e.g.*, text, audio, video and even *code*). While RNNs have demonstrated great performance [33] on a sequence prediction task with in-network memory, they struggle with processing long sequences because of the vanishing gradient problem [62]. To address this issue, gating models, such as LSTM (Long Short-Term Memory) [27] and GRU (Gated Recurrent Unit) [10], have been proposed by devising a special cell for long-range error propagation. However, they still suffer from i) limited capability of tracking long-term dependencies (*i.e.*, a single vector from an encoder that implies all previous words may lose partial information), and ii) disallowing parallelizable computation due to sequentiality. These shortcomings necessitate a better architecture for procssing binary functions, which often consist of multiple instructions.

C. ATTENTION AND TRANSFORMER

The Attention [3] mechanism in NLP considers all input words (at each time step) when predicting an output word, with a focus on a specific word associated with the output word for prediction. This approach captures the contextual relationship between words in a sentence without the gradient vanishing problem, which is widely adopted in a machine translation domain. Transformers [59] propose a *multi-head self-attention* technique for highly inferring the context of a sentence (a binary function for our purpose) atop the Attention's encoder-decoder architecture. Self-attention focuses on the inner relationship between input words, and multi-head considers multiple vectors with positional information to predict the next word (*i.e.*, instruction). Figure 2 (b) illustrates a single encoder layer in the Transformer architecture, which i)takes input vectors from the previous layer for computing an Attention matrix, ii) feeds the resulting vectors into a feed forward neural network (FFNN) in turn, iii) applies batch (*e.g.*, across training examples) and layer (*e.g.*, across feature dimensions) normalization between the self-attention and FFNN layers. The original Transformer [59] has six encoders (Figure 2a) and six decoders.

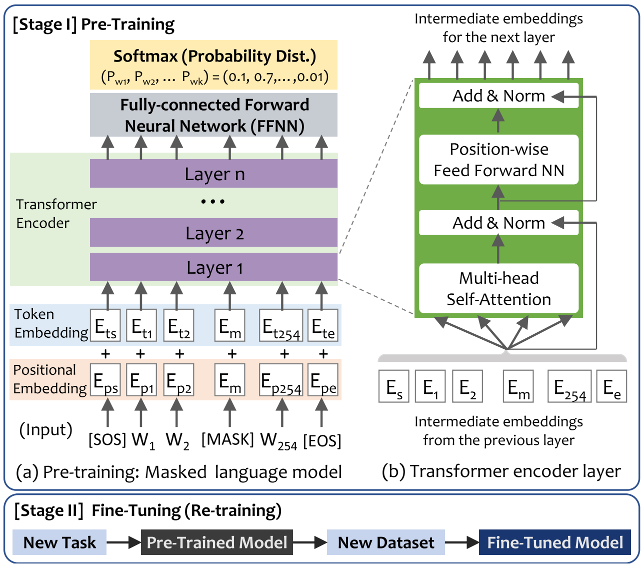

[Stage I] Pre-Training
Softmax (Probability Dist.)
$(P_{w1}, P_{w2}, \dots P_{wk}) = (0.1, 0.7, \dots, 0.01)$
Fully-connected Forward Neural Network (FFNN)
Transformer Encoder
Layer n
Layer 2
Layer 1
Token Embedding
E_{ts} E_{t1} E_{t2} E_m E_{t254} E_{te}
Positional Embedding
E_{ps} E_{p1} E_{p2} E_m E_{p254} E_{pe}
(Input)
[SOS] W_1 W_2 [MASK] W_{254} [EOS]
(a) Pre-training: Masked language model
Intermediate embeddings for the next layer
Add & Norm
Position-wise Feed Forward NN
Add & Norm
Multi-head Self-Attention
E_s E_1 E_2 E_m E_{254} E_e
Intermediate embeddings from the previous layer
(b) Transformer encoder layer
[Stage II] Fine-Tuning (Re-training)
New Task
Pre-Trained Model
New Dataset
Fine-Tuned Model

FIGURE 2. Simplified BERT structure. The key aspect of BERT is two trainings: a pre-training stage for generating a generic model, and a fine-tuning stage for a special model tailored to a downstream task. Note that pre-training requires a one-time but costly processing, whereas fine-tuning is computationally less expensive. `E` denotes an embedding per word `W`, and the subscripts `p`, `t`, `m`, `s`, `e`, and a number represent a position, token, `[MASK]`, `[SOS]`, `[EOS]`, and a word identifier, respectively.

D. LANGUAGE MODEL AND BERT

BERT [16], Bi-directional Encoder Representations from Transformers, is a state-of-the-art architecture that captures the contextual meaning of words and sentences in natural language by adopting the encoder layer of Transformer [59]. It incorporates several advanced concepts such as ELMo [50], semi-supervised sequence learning [12] and Transformer. BERT consists of two training phases as illustrated in Figure 2: a *pre-training* process that builds a generic model with a large corpus, and a *fine-tuning* process that updates the pre-trained model for a specific downstream task. The pre-training process employs two strategies: masked language model (MLM) and next sentence prediction (NSP), which are used for building a language model that considers context and the orders of words and sentences (through unsupervised learning with an unlabeled dataset). In Figure 2 (a), the `[MASK]` token represents an input word that has been masked, and `[SOS]` and `[EOS]` are tokens for the start and end of a sentence, respectively.[2] The `[UNK]` token is used for unknown words. In this example, a 256 fixed-length input (254 words with masked ones excluding two special tokens: start/end of a sentence at both ends) is used at a time. Once the pre-training is complete, the pre-trained model can be repurposed for varying other downstream tasks with supervised

[2] `[CLS]`, and `[SEP]` tokens in the original BERT correspond to our `[SOS]` and `[EOS]`.

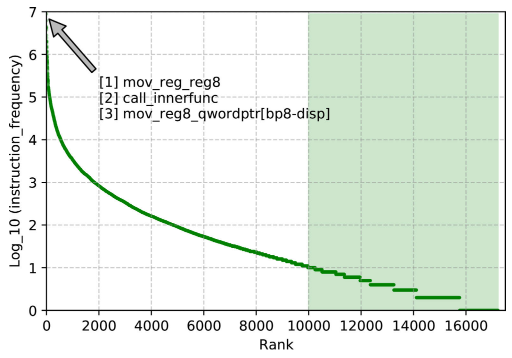


FIGURE 3. Log scale of instruction frequencies by its rank in our vocabulary corpus. The curve closely follows Zipf's law, akin to a natural language. The green area illustrates a long tail that has been rarely seen (*e.g.*, less than 10 times). Interested readers refer to Table 8 in Appendix.

learning. We adopt BERT because its structure seamlessly fits our objective of creating a pre-trained model that contains a generic binary code representation and re-training that model for a wide range of classification tasks with relatively lower computational resources (See Section IV-C for more details).

III. BINARY CODE SEMANTICS

In this section, we discuss the definition of code semantics, followed by highlighting a few insights of binary codes.

A. DEFINITION OF CODE SEMANTICS

Our approach considers binary representations of code semantics as distinct from those found in source code. In particular, the *equivalent semantics of a binary code* can be defined as a sequence of instructions that carries out the identical logical function as the original source. However, a binary function differs from a programmer-written function due to varying transformations by a compiler toolchain. To measure the similarity between two binary functions, we use a cosine similarity score that ranges from -1 to 1. A higher value indicates a closer relationship in code semantics.

B. OBSERVATIONS AND INSIGHTS

A binary code consists of a sequence of machine instructions that is analogous to a natural language. InnerEye [74] borrows ideas of Neural Machine Translation (NMT) to a binary function similarity comparison task by considering instructions as words and basic blocks as sentences. For successful binary code representation with deep neural networks, it is essential to have an in-depth understanding its properties. Here are several insights based on our analysis of machine instructions.

- **Machine instructions follow Zipf's Law.** Figure 3 depicts the relationship between the rank of instructions and the log scale of their frequencies. Our finding shows that the curve of the instruction distribution closely follows Zipf's law [73] like natural language, indicating that effective techniques in the NLP domain, such as BERT, can be useful binary tasks.
- **A function often conveys a meaningful context.** We analyzed 1, 681, 467 functions (18, 751, 933 basic blocks or 108, 466, 150 instructions) in our corpus (Table 3).[3] We measure several statistics: i) the number of instructions per function on average (I/F) is 64.5 (median=19, std=374.7), ii) the number of basic blocks per function on average (B/F) is 5.8 (median=4, std=16.4), and iii) the number of instructions per basic block on average (I/B) is 11.2 (median=3, std=95.8). As the standard deviation is quite large, To remove outliers, we cut off values larger than three times the standard deviation, resulting in a mean of (I/F, B/F, I/B) = (25.1, 3.9, 3.7). Figure 4 illustrates CDFs (upper) and histograms (lower) without outliers, indicating that approximately 70% of basic blocks contain five instructions or less. A binary function typically consists of around four basic blocks with 25 instructions on average, rendering a single function granularity sufficient to convey meaningful information. Indeed, our experimental results (Figure 1) with DeepBinDiff [67] show that quite a few matching blocks contain a couple of instructions (*e.g.*, `jmp`, `call`), which are highly likely to miss surrounding contexts.[4] We discuss other cases that a fine-grained granularity becomes beneficial (See section VII).
- Word2vec is unable to provide diverse representations for the same instruction in different contexts. A majority of prior works [17], [67], [74] adopt the Word2vec [44] algorithm to represent a binary code. Word2vec is an embedding technique that learns relationships between words in a large corpus of text, representing each distinct word with a vector. Figure 5 illustrates the Top 30 most common instructions that are well associated with each

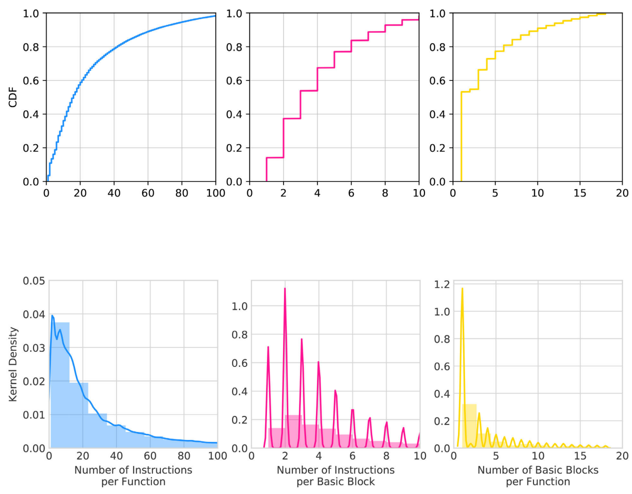


FIGURE 4. CDFs and histograms with kernel density estimates (10 bins) for the number of instructions per function (left) and basic block (middle), and the number of basic blocks per function (right) after removing outliers. Most basic blocks (around 80%) contain merely five instructions or less, indicating that a larger granularity (*e.g.*, function) is needed to imply contextually fruitful information.

[3]Note that we include user-defined functions; *e.g.*, linker-inserted functions are excluded.

[4]Instead, DeepBinDiff considers CFGs to read underlying context.

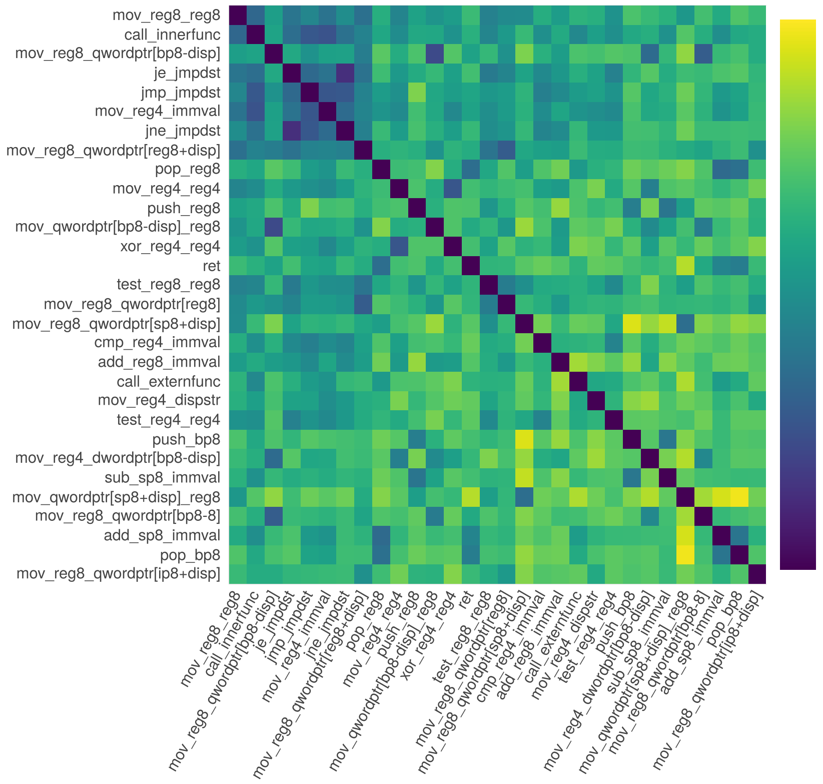


FIGURE 5. **Visualization of a similarity matrix for the most common 30 instructions with a Word2Vec model. The dark color represents that the relationships of how two instructions are close to each other. For example,** `ret` **(middle-left) comes with a series of function epilogue instructions (*e.g.,*** `pop bp8` **and** `add sp8 immval` **at the right-bottom corner). However, a fixed form of instruction vectorization is not sufficient because its location can have diverse meanings; *e.g.,*** `ret` **at the end and** `ret` **in the middle of a function.**

other. However, Word2vec cannot provide distinct representations for the same instruction in different contexts due to the absence of position information. For instance, the behavior of popping a register for a function epilogue differs from that for other computations in the middle of the function. Nevertheless, Word2vec assigns the identical representation (embedding) to the same word, regardless of its contextual differences, highlighting the need for a better embedding technique with a *restricted* number of vocabularies.

Unlike natural language, the number of possible instructions is virtually countless when each instruction is mapped into a single word (token or vocabulary); an immediate value in a 64-bit operand of the instruction could produce 2^{64} different words, making further computation impractical. To address this issue, most prior approaches that harness deep learning techniques [17], [67], [69], [74] *normalize instructions* before feeding a sequence of them as input into *the* training process. In particular, striking a balance is crucial so that instruction normalization can be neither too generic nor too specific because each token contains rich information while minimizing the OOV problem. Therefore, a better instruction normalization technique is needed to capture code semantics for neural networks.

IV. DeepSemantic DESIGN

In this section, we introduce a better instruction normalization technique, and portray DeepSemantic to show its effectiveness in detail.

A. WELL-BALANCED INSTRUCTION NORMALIZATION

An efficient instruction normalization process is crucial to prepare its vectorization form for a neural network as adopted by many prior approaches [17], [43], [67], [69], [74]. In other words, the final contextual information is determined by the quality of word embeddings based on an individual normalized instruction. However, a too-coarse-grained normalization, such as stripping all immediate values [17], [67], [74], loses a considerable amount of contextual information whereas a too-fine-grained normalization (close to the original instruction disassembly) raises OOV due to a massive number of unseen instructions (tokens). We observe that the previous approaches sorely perform mechanical conversion of either an opcode or operand(s) without thorough consideration of their contextual meanings. Figure 7 shows different normalization strategies taken by several approaches for binary similarity detection. DeepBinDiff [67] considers a register size to symbolize an n-byte register (❸); however, it

TABLE 1. Summary of well-balanced instruction normalization (WIN) rules that target an operand for `x86_64`: immediates, registers and pointers. Our strategy aims to remain as much contextual information as possible for further embedding generation.

Operand	Rule	Description or Expression	Notation	Note
Immediate	call target	`libc` library call	`libc`[name]	points to a libc function
		recursive call	self	points to a function itself
		function call within a binary	innerfunc	points to a `.text` section
		function call out of a binary	externfunc	points to a `.got` or `.plt` section
	jump family	destination to jump to	jmpdst	destination within a function
	reference	string	dispstr	refers to a string
		statically allocated variables	dispbss	points to a `.bss` section
		data	dispdata	refers to data other than a string
	default	all other immediate values	immval	all cases other than the above
Register	size	[e|r]*[a|b|c|d|si|di][x|l|h]*, r[8-15][b|w|d]*	reg[1|2|4|8]	size information
	stack/base/instruction	[e|r]*[b|s|i]p[l]*	[s|b|i]p[1|2|4|8]	special registers (*e.g.*, stack)
	special purpose	cr[0-15], dr[0-15], st([0-7]), [c|d|e|f|g|s]s	reg[cr|dr|st], reg[c|d|e|f|s]s	special registers (*e.g.*, flags)
	special purpose (avx)	[x|y|z]*mm[0-7|0-31]	reg[x|y|z]*mm	Advanced vector extensions registers
Pointer	direct (small size)	byte,word,dword,qword,ptr	memptr[1|2|4|8]	pointer with a small size ($\leq$ 8 bytes)
	direct (large size)	tbyte,xword,[x|y|z]mmword	memptr[10|16|32|64]	pointer with a large size ($>$ 8 bytes)
	indirect (string)	[base+index*scale+displacement]	[base+index*scale+dispstr]	pointer that refers to a string
	indirect (others)	[base+index*scale+displacement]	[base+index*scale+disp]	pointer with a displacement

converts all immediates into `imme` (❾). Meanwhile, InnerEye [74] discards the size information of registers (❿) for a 64-bit machine instruction set. SAFE [43] retains immediate values (❼). Besides, all three cases convert the destination of a call invocation into a single notation (*e.g.*, `HIMM`, `imme`, or `FOO`), rendering every call instruction identical.

To this end, we introduce a *well-balanced instruction normalization* (WIN) strategy that strikes a balance between the expressiveness of binary code semantics and a reasonable amount of tokens, preserving the semantics of instructions. For example, we consider various implications for immediate values such as library targets, call invocations, destinations to jump to, string references, and statically-allocated variables. As another example, our experiments show that the two most frequent words, `mov_reg8_ptr` and `mov_reg8_reg8`, which account for over 20% of appearances with coarse-grained normalization, cannot convey a valid context. Our approach enhances the quality of word embeddings to capture the semantic nuances of a code snippet.

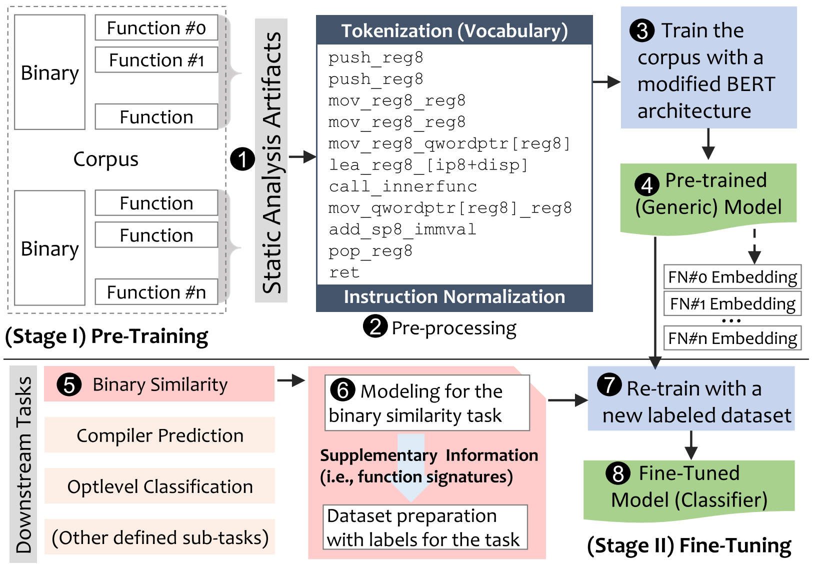
Table 1 summarizes three basic principles in mind to balance seemingly conflicting goals above: i) an immediate can fall into a jump or call destination (*e.g.*, ❷ 0×401`d00` $\rightarrow$ `externfunc`, ❻ 0×425530 $\rightarrow$ `innerfunc`), a value itself (*e.g.*, ❺ 0×38 $\rightarrow$ `immval`) or a reference (*e.g.*, ❹ $0 \times 425530 \rightarrow$ `dispbss`), according to a string literal, statically allocated variable or other data; ii) a register can be classified with a size by default (*e.g.*, ❶ `r14` $\rightarrow$ `reg8`, ❹ `eax` $\rightarrow$ `reg4`); but the ones with a special purpose stay intact such as a stack pointer, instruction pointer, or base pointer (*e.g.*, ❸ `ebp` $\rightarrow$ `bp4`); and iii) a pointer expression follows the original format, "`base+index*scale+displacement`" (*e.g.*, ❸ `DWORD PTR [r14]` $\rightarrow$ `dwordptr[reg8]`) so that certain memory access information can be preserved; moreover, the same rule applies if and only if the displacement refers to a string reference (*e.g.*, `dispstr`). Note that an opcode is not part of our normalization process.

B. OVERVIEW

Equipped with WIN and our insights on binary code, we present DeepSemantic atop BERT. Adopting the original BERT scheme, DeepSemantic consists of two separate stages (Figure 6): i) a pre-training stage that creates a general model applicable to a downstream task, and ii) a fine-tuning stage that generates a special model for a downstream task on top of the pre-trained model. In this paper, we select a binary code similarity task because its performance predominately relies on the inference of code semantics. The following justifies our design choices behind DeepSemantic :

- **Function-level Granularity.** We determine a function as a basic unit that can imply meaningful semantics from our insights in Section III-B. Indeed, our experiment shows a large portion (*e.g.*, 83.25%) of basic block matching results with the previous approach [67] come from very small basic blocks (Figure 1).
- **Function Embedding.** Along with a function-level granularity, DeepSemantic generates an embedding per function as a whole, rather than per instruction (*e.g.*, word2vec) to represent a code snippet. Besides, an identical instruction would have a different embedding depending on its position and surrounding instructions (Section 2).
- **Well-balanced Instruction Normalization.** We leverage the existing static binary analysis to normalize instructions so that a pre-training model can naturally embrace important features in a deep neural network. We intentionally attempt to remain as much information (manually engineered features from previous studies [18]) as possible (Section IV-A).

As DeepSemantic inherently generates multiple models, the pre-training and fine-tuning models are dubbed DS-Pre and DS-Task, respectively, depending on the task. In this work, we build DS-BinSim for binary code similarity detection as a downstream task.



Binary
Function #0
Function #1
Function
Corpus
Binary
Function
Function
Function #n
❶
Static Analysis Artifacts
Tokenization (Vocabulary)
Instruction Normalization
❷ Pre-processing
(Stage I) Pre-Training
❸ Train the corpus with a modified BERT architecture
❹ Pre-trained (Generic) Model
FN#0 Embedding
FN#1 Embedding
FN#n Embedding
Downstream Tasks
❺ Binary Similarity
Compiler Prediction
Optlevel Classification
(Other defined sub-tasks)
❻ Modeling for the binary similarity task
Supplementary Information (i.e., function signatures)
Dataset preparation with labels for the task
❼ Re-train with a new labeled dataset
❽ Fine-Tuned Model (Classifier)
(Stage II) Fine-Tuning

```
push_reg8
push_reg8
mov_reg8_reg8
mov_reg8_reg8
mov_reg8_qwordptr[reg8]
lea_reg8_[ip8+disp]
call_innerfunc
mov_qwordptr[reg8]_reg8
add_sp8_immval
pop_reg8
ret
```

FIGURE 6. DeepSemantic Overview. For a pre-training phase, we use static analysis to collect artifacts from a pre-defined binary corpus ed❶. In a pre-processing step, we tokenize all instructions and normalize them ed❷ (Section IV-A). The entire corpus is then pre-trained with BERT ed❸, resulting in a pre-trained model ed❹ (Section IV-C) For a fine-tuning phase, we define a downstream task, such as binary similarity comparison ed❺, and prepare a dataset with appropriate labels ed❻. Finally, after re-training on the downstream task ed❼, we obtain a fine-tuned model ed❽ (Section IV-D).

Original Disassembly	DeepSemantic	SAFE	DeepBinDiff	InnerEye
push rbp	push_bp8/	X_push_rbp/	push/reg8/	PUSHQ~RBP/
push r14 ❶	**push_reg8/**	**X_push_r14/**	**push/reg8/** ❽	**PUSHQ~R14/**
push rbx	push_reg8/	X_push_rbx/	push/reg8/	PUSHQ~RBX/
mov rbx,rdi	mov_reg8_reg8/	X_mov_r11d,_r15d/	mov/reg8/reg8/	MOVQ~RBX,RDI/
call 401d00 <fn1@plt> ❷	**call_externfunc/**	**X_call_HIMM/**	**call/imme/**	**CALLQ~FOO/**
mov r14,rax	mov_reg8_reg8/	X_mov_esi,_eax/	mov/reg8/reg8/	MOVQ~R14,RAX/
mov ebp,DWORD PTR [r14] ❸	**mov_bp4_dwordptr[reg8]/**	**X_mov_ebp,_[r14*1+0]/**	**mov/reg4/ptr/**	**MOVQ~RBP,[R14]** ❿
test rbx,rbx	test_reg8_reg8/	X_test_rbx,_rbx/	test/reg8/reg8/	TESTQ~RBX,RBX/
mov eax,0x425530 ❹	**mov_reg4_dispbss/**	**X_mov_eax,_HIMM/**	**mov/eax/imme/** ❾	**MOVQ~RAX,0/**
cmove rbx,rax	cmove_reg8_reg8/	X_cmove_rbx,_rax/	cmove/rbx/rax	CMOVEQ~RBX,RAX/
mov esi,0x38 ❺	**mov_reg4_immval/**	**X_mov_esi,_0x38/** ❼	**mov/reg4/imme/**	**MOVQ~RSI,0/**
mov rdi,rbx	mov_reg8_reg8/	X_mov_rdi,_rbx/	mov/reg8/reg8	MOVQ~RDI,RBX/
call 40a130 <fn2> ❻	**call_innerfunc/**	**X_call_HIMM/**	**call/imme/**	**CALLQ~FOO/**
...	...	...	...	...

FIGURE 7. Examples of the proposed WIN (Table 1) prior to code embedding generation. A slash (/) represents a token separator; *e.g.*, In DeepBinDiff [67] opcode and operand(s) are separate tokens. Note that too-coarse-grained normalization (*e.g.*, DeepBinDiff [67], InnerEye [74]) may lose contextual information; whereas too-fine-grained (*e.g.*, SAFE [43]) normalization may suffer from OOV. We explain each case (❶-❿) in Section IV-A.

C. GENERIC MODEL FOR ASSEMBLY CODES

For NLP applications, it's often feasible to employ a (readily available) pre-trained BERT model on a large corpus of human words (*e.g.*, from Wikipedia) because BERT is designed to be computationally expensive and versatile for various downstream task. However, we cannot utilize such a model for a human language. Thus, training DeepSemanticon a new dataset with a different vocabulary, such as machine instructions in our corpus, is necessary. We adopt the original BERT's masked language model (MLM), which probabilistically masks a pre-defined portion of normalized instructions (*e.g.*, 15%), and then predicts them within a given function during pre-training. It is noteworthy mentioning that DeepSemantic does not use NSP (*i.e.*, prediction of next sentence) because two consecutive functions often are not semantically connected. Besides, our model takes advantage of the Transformer architecture, which allows for direct connection between all instructions efficiently and highly parallelizable computation (*e.g.*, GPU resources).

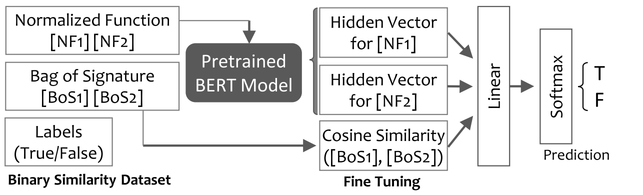


FIGURE 8. Binary similarity prediction model (DS-BinSim) as a fine tuning task. We concatenate two hidden vectors from the two normalized functions (NFs) and the cosine similarity of the two Bag of Signatures (BoSes). Note that BoS offers supplementary information (susubsection IV-D1).

D. SPECIAL MODEL FOR A DOWNSTREAM TASK

DeepSemantic aims to support specific downstream tasks that require the inference of contextual information from binary code, and it achieves this through quick re-training based on a pre-trained model that provides generic code representation. During fine-tuning, a separate dataset with labeled data is required for supervised learning, adjusting a generic model tailored to a specialized task. In this paper, we focus on demonstrating the effectiveness of our model for a single downstream task: binary similarity (DS-BinSim) that predicts the similarity of two functions.

1) DS-BinSim MODEL

a: BINARY CODE SIMILARITY TASK

We define our downstream task as the estimation of similarity between two binary functions that originate from the same source code. To accomplish this, we create a new dataset that comprises pairs of normalized functions (NFs) with a binary label whether the functions are identical. Figure 8 illustrates our binary similarity model as a downstream task. Our model is built on top of DS-Pre, using it as a basis to obtain two hidden vectors (of size h) from each NF.

b: BAG OF SIGNATURE (BoS)

We provide supplementary information, dubbed *Bag of Signature (BoS)*.[5] to enhance the binary similarity task. The idea behind BoS is that even a WIN process (Table 1) loses the information of a string or numeric constant. In essence, we enumerate both string literals and numeric constants from a static analysis phase, combining them into a single bag that can be used as a unique signature afterwards. We compute a BoS similarity score with Equation 1.

$$BoS(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}||\vec{w}|} = \frac{\sum_{i=1}^{n} v_i w_i}{\sqrt{\sum_{i=1}^{n} v_i^2}\sqrt{\sum_{i=1}^{n} w_i^2}} \quad (1)$$

As a concrete example, a function (F) contains a list of `[1, 0×12, 8, 8, 8, 'Hello']` where function (G) holds `[0×12, 8, 8, 'Hello']`. Then, we can represent each vector based on counting those constants: $F = (1, 1, 3, 1)$ and $G = (0, 1, 2, 1)$ whose cosine similarity becomes 0.943.

c: VECTOR CONCATENATION

We concatenate three vectors including the two hidden vectors for the functions and the cosine similarity of two BoSes, passing them through a linear layer with inputs $2 * h + 1$.

2) MODEL AND LOSS FUNCTION

For a binary similarity detection task, the logits can be calculated as following:

$$\hat{y} = \text{Softmax}(\mathcal{F}(h)) \quad (2)$$

where $\mathcal{F}(\cdot)$ and h are a fully-connected layer and the hidden vector of the given function returned from DS-Pre, respectively. To obtain the optimal network parameters in the fine-tuning layers, we use cross-entropy as a loss function; seeking the network parameters θ that satisfy:

$$\theta = \arg\min_{\theta} \sum_{c \in C} p(c|y) log(p(c|\hat{y})) \quad (3)$$

where C, $p(c|y)$, and $p(c|\hat{y})$ denote a set of classes (*e.g.*, decision of function similarity in DS-BinSim), the ground-truth class distribution, and the estimated probability for the class c by the logits $\hat{y}$ from Equation 2.

[5]Prior approaches [9], [18], [58] reveal that a string or numeric constant can be an important feature vector. The improvement of DeepSemantic with additional information is marginal (0.09% as in Figure 8).

V. IMPLEMENTATION

This section briefly describes the artifacts from static binary analysis and DeepSemantic implementation.

A. BINARY ANALYSIS ARTIFACTS

With our corpus (Table 3), we extract essential artifacts from one of the state-of-the-art static binary analysis tools, IDA Pro [24]. We leverage `IDAPython` [25] (a built-in IDA Pro [24] plugin) to build an initial database of binary analysis artifacts including function names, `libc` library calls, cross references (*e.g.*, string literals, numeric constants), section names and call invocations (*e.g.*, internal calls, external calls), which further assists the WIN process (Table 1). Although we provide a binary with debugging symbols available to confirm the ground truth (*e.g.*, function boundaries) during static analysis, it is noted that DeepSemanticdoes not require such debugging information. Any binary analysis tool would suffice to recognize binary functions such as angr [2], Ghidra [46], or radare2 [54].

B. BERT HYPERPARAMETERS

We develop DeepSemantic with Tensorflow [22] and PyTorch [53] on top of existing BERT implementations [23], [30], [63]. As described in section IV, we have DeepSemantic exclude NSP when building a language model from the original BERT architecture because the semantic of a binary function is agnostic to that of adjacent functions. We pre-train with a batch size of 96 sequences, where each sequence contains 256 tokens (*e.g.*, 256 * 96 = 24,576 tokens/batch including special tokens) using five epochs over the 1.3M binary functions. We use the Adam optimizer with a learning rate of 0.0005, $\beta_1 = 0.9$, $\beta_2 = 0.999$, an L2 weight decay rate of 0.01 with a liner decay of the learning rate. We use a dropout rate of 0.1 on all layers, and the `ReLU` activation function. Table 2 encapsulates all hyperparameters when we build our models for the BERT language model, optimizer, and trainer. The number of trainable parameters is 8, 723, 914 for DS-Pre.

VI. EVALUATION

Since the approach of DeepSemantic involves a training process twice, DS-Pre and DS-Task, we set up various experiments to ensure accurate and fair evaluation. We first build a DS-Pre model (Section VI-C), and then answer the follow-

TABLE 2. Hyperparameters for (B)ERT, (O)ptimizer, and (T)rainer during a training phase.

(B) Dimension of embeddings	256
(B) Number of hidden layers	128
(B) Number of attention layers	8
(B) Number of attention heads	8
(B) Maxium length of encoding	250
(B) Position dropout	0.1
(B) Conv1d dropout	0.2
(B) Number of conv1d layers	3
(B) Size of conv1d kernel	5
(B) Feed forward network dropout	0.1
(B) Self-attention dropout	0.1
(O) Loss rate	0.0005
(O) Adam beta1	0.9
(O) Adam beta2	0.999
(O) Adam weight decay	0.01
(O) Adam epsilon	1.00E-06
(O) Warmup	Linear
(T) Epochs	5
(T) Batch size	96
(T) Batch size	96
(T) Train dataset ratio	0.9
(T) Valid dataset ratio	0.05
(T) Test dataset ratio	0.05

ing four research questions pertaining to effectiveness and efficiency.

- **RQ1.** To what extent does DS-BinSim outperform existing cutting-edge approaches, such as DeepBinDiff [67], SAFE [43], for a binary similarity detection task that requires the inference of underlying binary semantics (Section VI-D)?
- **RQ2.** How well does a fine-tuning task enhance binary code representation? We assess how DS-BinSim updates the original function embedding vectors from DS-Pre (Section VI-E).
- **RQ3.** To what extent does WIN offer improvements for the DS-BinSim model? (Section VI-F)
- **RQ4.** How efficient is DeepSemantic in practice (Section VI-G)?

A. ENVIRONMENT

We evaluate DeepSemantic on a 64-bit Ubuntu 18.04 system equipped with an Intel(R) i9-10900X CPU (with 20 3.70 GHz cores), 128 GB RAM and an NVIDIA Quadro RTX 8000 GPU.

B. DATASET

Our dataset includes the entire corpus described in Table 3. We generated 1,328 binaries that were compiled with two compilers (*e.g.*, `gcc` 5.4 and `clang` 6.0.1) and four optimization levels (*e.g.*, `O[0-3]`) from three different test-suites (*e.g.*, GNUtils, SPEC2017, the utilities including `openssl` [47], `nginx` [45], and `vsftpd` [61]. Additionally, the 176 SPEC2006 binaries are borrowed from the official release [49].

C. DS-Pre GENERATION

We pre-train DS-Pre model with the artifacts corresponding to each binary from Section V. We first normalize all

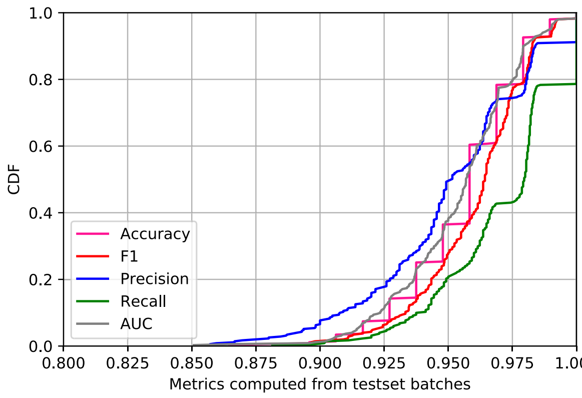


FIGURE 9. CDF of varying metrics to show the performance of DS-BinSim (See Table 6).

instructions, define tokens, and create a dataset for BERT pre-training. In total, we obtain 17,225 tokens from all 107.88 million instructions in our corpus, including each (normalized) instruction token five special tokens (`[SOS]`: start of a sentence, `[EOS]`: end of a sentence, `[UNK]`: unknown token, `[MASK]`: mask to predict a word, `[PAD]`: padding symbol to fill out an input length) for DS-Pre generation, a modified BERT model as described in Figure 2. Our corpus contains 1,690,715 normalized binary functions in total, however, we solely include around 1.3 million functions after filtering out functions that are too small (*i.e.*, those with five or fewer, which account for 15.4% of the dataset) or too large (*i.e.*, those with more than 250 instructions, which account for 4.5% of the dataset) ones. The small functions are excluded because they do not provide enough meaningful semantic chunks for training, and the large functions may hinder the performance of the pre-training model generation. However, we have not excluded identical NFs during pre-training model generation because they can be treated as different training data due to the probabilistic masking mechanism in MLM. During testing, the out-of-vocabulary (OOV) ratio is around 0.93%, with only 82 vocabularies being unknown out of 8,783 tokens (the number of vocabularies in the training set is 17,024).

D. EFFECTIVENESS OF DS-BinSim

This section demonstrates the effectiveness of DS-BinSim for a binary similarity task by comparing it with the two state-of-the-art deep learning tools. As illustrated in Figure 8, we prepare a dataset that consists of pairs of two normalized functions (NFs), and their corresponding labels. The label is 1 (true) if two binary functions come from the same source code, and 0 (false) otherwise. With the same hyperparameters in Table 2, we fine-tune the model (DS-BinSim) to predict binary similarity. Figure 9 illustrates the CDF of varying metrics, including an F1 of 0.965 and an AUC of 0.959 on average (See other metrics in the first column in Table 6). This means our classifier can accurately predict whether two binary functions are compiled from the same source,

TABLE 3. Summary of our whole corpus. FN, BB and IN represent a function, basic block and instruction, respectively. A small function indicates the number of instructions per function (I/F) is less than or equal to 5, whereas a large function indicates I/F is greater than 250. We also investigate the number of immediate operands, string references and `libc` call invocations per function to devise well-balanced instruction normalization (WIN).

	Number of Occurrence					Rate		Metrics			Average Occurrence per FN		
Testsuite	**Binaries**	**FNs**	**BBs**	**INs**	**Vocas**	**Small FNs**	**Large FNs**	**BBs/FN**	**INs/FN**	**INs/BB**	**Immediate**	**String**	**Libc**
GNUtils	1,000	446,014	3,983,384	22,216,363	5,328	13.70%	3.14%	8.93	49.81	5.58	7.99	1.01	0.88
Spec2006	176	408,496	4,664,973	28,321,431	9,632	14.96%	5.01%	11.42	69.33	6.07	9.72	0.81	0.41
Spec2017	120	755,673	9,339,087	52,952,315	11,933	13.90%	4.83%	12.36	70.07	5.67	10.37	0.68	0.28
Utilities	32	80,532	788,861	5,083,417	4,963	18.89%	4.94%	9.80	63.12	6.44	11.95	0.97	0.25
Total	1,328	1,690,715	18,776,305	108,573,526	17,220	15.36%	4.48%	10.63	63.08	5.94	10.01	0.87	0.46

TABLE 4. Precision (P), Recall (R), and F1 results of binary similarity comparison across different compiler and optimization level pairs. The leftmost column represents a pair where C, G, O[0-3] denote clang, gcc, and an optimization level, respectively. We conduct two separate experiments: DeepBinDiff [67] and DS-BinSim with a limited dataset (left), and SAFE [43] and DS-BinSim with a full dataset (right). This is because DeepBinDiff is incapable of computing binary similarity scores for a certain set of binaries. DS-BinSim outperforms DeepBinDiff (*e.g.*, around 49.8%) by a large margin, and SAFE (*e.g.*, 15.8%) across all pair combinations.

Dataset	Limited Testset							Full Testset						
Comparison	**DeepBinDiff**			**DS-BinSim**				**SAFE**			**DS-BinSim**			
Pair	**P**	**R**	**F1**	**P**	**R**	**F1**	**Diff**	**P**	**R**	**F1**	**P**	**R**	**F1**	**Diff**
(CO0, CO1)	0.574	0.197	0.293	0.907	0.907	0.907	61.40%	0.879	0.728	0.796	0.958	0.975	0.967	17.03%
(CO0, CO2)	0.528	0.176	0.264	0.924	0.910	0.917	65.29%	0.774	0.676	0.722	0.973	0.967	0.970	24.79%
(CO0, CO3)	0.501	0.178	0.263	0.910	0.924	0.917	65.43%	0.830	0.627	0.715	0.958	0.969	0.963	24.87%
(CO0, GO0)	0.693	0.279	0.398	0.921	0.930	0.925	52.72%	0.788	0.996	0.880	0.955	0.979	0.967	8.68%
(CO0, GO1)	0.529	0.173	0.261	0.934	0.915	0.925	66.40%	0.836	0.704	0.764	0.943	0.968	0.956	16.05%
(CO0, GO2)	0.506	0.151	**0.233**	0.930	0.920	**0.925**	**69.22%**	0.698	0.662	0.679	0.940	0.975	0.957	27.79%
(CO0, GO3)	0.488	0.151	0.231	0.932	0.905	0.918	68.74%	0.713	0.648	0.679	0.945	0.983	0.963	28.44%
(CO1, CO2)	0.727	0.643	0.682	0.928	0.921	0.924	24.20%	0.770	0.824	0.796	0.954	0.973	0.963	16.70%
(CO1, CO3)	0.705	0.612	0.655	0.930	0.911	0.921	26.55%	0.685	0.824	0.748	0.963	0.989	0.976	22.78%
(CO1, GO0)	0.599	0.202	0.302	0.931	0.917	0.924	62.15%	0.796	0.794	0.795	0.954	0.981	0.967	17.22%
(CO1, GO1)	0.613	0.330	0.429	0.923	0.909	0.916	48.67%	0.871	0.955	0.911	0.942	0.974	0.958	4.69%
(CO1, GO2)	0.592	0.375	0.459	0.914	0.911	0.913	45.33%	0.737	0.882	0.803	0.954	0.984	0.969	16.55%
(CO1, GO3)	0.546	0.346	0.424	0.921	0.915	0.918	49.41%	0.732	0.909	0.811	0.949	0.960	0.955	14.37%
(CO2, CO3)	0.947	0.863	0.903	0.925	0.917	0.921	1.79%	0.662	0.978	0.789	0.959	0.971	0.965	17.55%
(CO2, GO0)	0.509	0.187	0.274	0.933	0.918	0.926	65.21%	0.679	0.689	**0.684**	0.951	0.981	**0.965**	**28.14%**
(CO2, GO1)	0.600	0.302	0.402	0.919	0.903	0.911	50.88%	0.813	0.952	0.877	0.968	0.982	0.975	9.75%
(CO2, GO2)	0.606	0.343	0.438	0.918	0.918	0.918	47.99%	0.825	0.942	0.880	0.965	0.986	0.975	9.54%
(CO2, GO3)	0.598	0.344	0.437	0.934	0.913	0.923	48.65%	0.830	0.949	0.885	0.939	0.978	0.958	7.24%
(CO3, GO0)	0.494	0.179	0.263	0.910	0.910	0.910	64.73%	0.731	0.746	0.738	0.945	0.990	0.967	22.85%
(CO3, GO1)	0.577	0.300	0.395	0.937	0.930	0.934	53.87%	0.864	0.941	0.901	0.950	0.961	0.955	5.42%
(CO3, GO2)	0.605	0.349	0.443	0.928	0.929	0.929	48.60%	0.909	0.961	0.934	0.948	0.977	0.962	2.81%
(CO3, GO3)	0.579	0.342	0.430	0.920	0.920	0.920	49.01%	0.791	0.966	0.870	0.947	0.962	0.954	8.45%
(GO0, GO1)	0.582	0.221	0.320	0.916	0.910	0.913	59.24%	0.757	0.802	0.779	0.964	0.967	0.966	18.66%
(GO0, GO2)	0.555	0.190	0.283	0.943	0.916	0.929	64.59%	0.815	0.687	0.745	0.953	0.962	0.958	21.20%
(GO0, GO3)	0.515	0.187	0.274	0.920	0.916	0.918	64.35%	0.796	0.651	0.716	0.955	0.982	0.968	25.21%
(GO1, GO2)	0.784	0.556	0.651	0.930	0.927	0.929	27.80%	0.815	0.993	0.895	0.961	0.967	0.964	6.92%
(GO1, GO3)	0.751	0.536	0.626	0.924	0.900	0.912	28.62%	0.854	0.944	0.897	0.961	0.969	0.965	6.80%
(GO2, GO3)	0.848	0.737	0.789	0.942	0.929	0.935	14.65%	0.755	0.938	0.837	0.949	0.982	0.965	12.85%
Average	0.613	0.337	**0.422**	0.925	0.916	**0.921**	**49.84%**	0.786	0.835	**0.805**	0.954	0.975	**0.964**	**15.83%**

regardless of a wide range of code transformations from arbitrary combinations of compilers and optimization levels.

1) COMPARISON WITH DeepBinDiff

We conducted an experiment to compare DeepSemantic with open-sourced DeepBinDiff [70]. Unfortunately, the officially released DeepBinDiff was not able to handle even medium-size binaries, so we defined a limited dataset consisting of 96 executables, including part of the SPEC2006 and findutils. [6] By design, DeepBinDiff performs basic block matching by taking two binary inputs, whereas DS-BinSim aims for comparisons at the function granularity. We classify each case ias a true positive for a fair comparison with DeepBinDiff when it discovers *any pair of matching basic blocks* that belong to the same function (relaxed judgment). Notably, we extended the evaluation of DeepBinDiff to different compilers (*i.e.*, gcc VS clang) and optimization levels (*i.e.*, 28 different combinations) to demonstrate the effectiveness of DS-BinSim. Table 4 shows that our approach considerably outperforms DeepBinDiff (49.8%) in all combinations of compiler and optimization levels. Our experiment aligns with the results across optimizations from DeepBinDiff [67].

2) COMPARISON WITH SAFE

We conducted another experiment with the full test dataset in our corpus to compare our approach with SAFE [43]. By design, SAFE merely computes a cosine similarity value does not make a decision on whether two functions are simliar or not. Hence, we choose a threshold of 0.5 (*i.e.*, if the value is greater than the threshold, the decision is true), and create a database for our whole dataset(Table 3) to query function embeddings with the open-sourced version of SAFE [41].

[6] `astar`, `bzip2`, `hmmer`, `lbm`, `libquantum`, `mcf`, `milc`, `namd`, `sphinx_livepretend`, and findutils (`find`, `xargs`, `locate`).

TABLE 5. Cosine similarity of similar/dissimilar function pairs on average between DS-Pre(pre-trained model) and DS-BinSim(fine-tuned model).

Category	DS-Pre	DS-BinSim
Similar	0.647	**0.751**
Dissimilar	0.273	**0.309**

TABLE 6. Various metrics to show the effectiveness of well-balanced instruction normalization in comparison with coarse-grained instruction normalization.

Metric	Normalization Granularity		
	Balanced	Coarse	(Diff)
Accuracy	**0.961**	0.934	2.712%
Precision	**0.955**	0.932	2.323%
Recall	**0.975**	0.962	1.347%
F1	**0.965**	0.946	1.869%
AUC	**0.959**	0.926	3.299%

Our approach outperformed SAFE by 15.8% on the full testset, as shown in Table 4. In particular, we observe a substantial difference in performance when code semantics are difficult to infer, such as when there is a high difference in optimization level.

Answer to RQ1. DS-BinSim by far outperforms DeepBinDiff (50% on average; up to 69%) and SAFE (16% on average; up to 28%) across all 28 combinations of different compilers and optimization levels.

E. CODE REPRESENTATION WITH FINE-TUNING

We evaluate the effectiveness of a 256-dimensional embedding (as shown in Table 2) that represents a function after fine-tuning. It is worth noting that there are several ways to extract a contextualized embedding [1], and in this study, we use the mean of all hidden states. To this end, we extract 179,163 unique pairs of functions from our corpus, with approximately half being similar, and other half dissimilar. We compute cosine similarity scores, excluding cases where two NFs are exactly identical because the score is always 1 in such cases. For example, the similarity score for the function embedding pair, `ngx_resolver_resend_handler` between the clang O1 (70 instructions) and gcc O2 (86 instructions) is 0.805 using the DS-Pre model. After fine-tuning with the DS-BinSim model, we obtain an improved similarity score of 0.826 for the above example, indicating that the vectorized values for the similar pair are close to 1. Table 5 briefly shows the average cosine similarity values, and we observe that the binary function representation has been enhanced; *i.e.*, the difference between the similar pairs in a vector space becomes broader on average.

Answer to RQ2. Our empirical results indicate that a fine-tuning process successfully enhances code representations for a specialized downstream task (*e.g.*, DS-BinSim).

F. EFFECTIVENESS of WIN FOR BINARY SIMILARITY

This section depicts how WIN with pre-defined rules (Table 1) enhances DS-BinSim. To assess its effectiveness, we compare DeepSemantic with a fine-grained model without normalization. In this case, only the rule of replacing

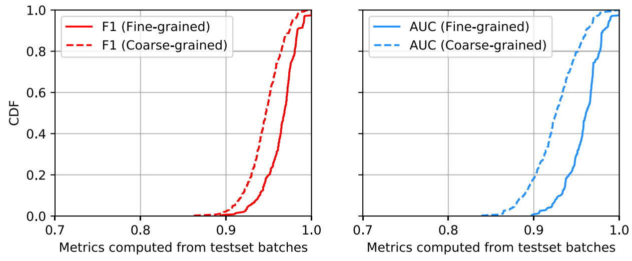


FIGURE 10. CDF comparison of F1 and AUC metrics between coarse-grained (dotted lines) and well-balanced instruction normalization (solid lines).

every immediate operand with `immval` is applied. Our training set has 4, 917, 904 tokens, which is 207.4 times larger than the WIN set. Training this model requires 283.6 GB GPU memory to update 1.38 billion trainable parameters. Putting large resource consumption aside, the problem would be exacerbated because most of the vocabularies (3, 716, 287 or 75.6%) appear only once, making it impossible to update corresponding instruction vectors during training. Moreover, in the test set, 97.8% (127, 805 out of 130, 736) of all tokens are not present in the training set, rendering further learning pointless due to a severe OOV problem (*i.e.*, most of vocabularies would be regarded as `UNK`).

To address this issue, we re-define the normalization rule to be relatively coarse-grained, replacing, all immediates and pointers with `immval` and `ptr` without taking any information into account except for registers. We generate a new dataset with 1, 174, 060 functions, reducing the number of tokens up to 2, 022, including five special ones, from 17, 225 of the WIN set (around 88.3% reduction). We generate another DS-Pre model for coarse-grained normalization with 2, 870, 759 trainable parameters. Then, we prepare another DS-BinSim dataset with labels using 100K pairs (the ratio of similar and dissimilar labels is 1:1) from our corpus. It is worth noting that we exclude all identical NF pairs for similar pairs (*e.g.*, all pairs must have at least one or more discrepancies) to compute a robust model and use negative sampling for dissimilar pairs.

As a result, we obtain higher F1 and AUC values of (0.965, 0.959) with WIN than with coarse-grained normalization, which yield F1 and AUC values of (0.946, 0.926) (See Table 6). Figure 10 shows the results with WIN in a 1.87% higher F1 and a 3.30% higher AUC. Notably, WIN reduces the false positive rate by 5.25%. Although the improvements are not significant, we believe that this approach may be beneficial for other contextually sensitive downstream tasks.

Answer to RQ3. We demonstrate WIN enhances code representation by providing supplementary information. We experimentally confirm that the normalization process is crucial for effective learning of semantic-aware code representation.

G. EFFICIENCY OF DeepSemantic

In this section, we exhibit the practical efficiency of DeepSemantic. Table 7 provides a concise summary of the

TABLE 7. Comparison of computational resources between training and testing for DS-BinSim.

Metrics	DS-PRE	DS-BINSIM-Training	DS-BINSIM-Testing
Per Epoch (secs)	3553.11	2229.10	43.52
Per Batch (secs)	0.32	0.35	0.12
GPU (GB)	34.75	37.27	37.27

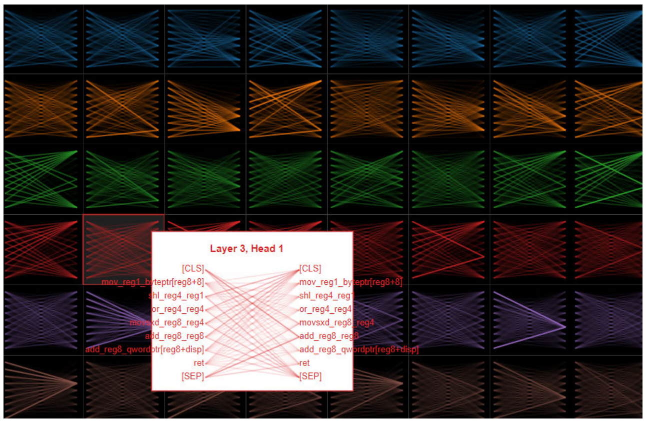


FIGURE 11. Visualization of a multi-head self-attention architecture in Transformer with BERTViz [60]. Note that [CLS] and [SEP] correspond to [SOS] and [EOS] in DeepSemantic.

computational resource required on average, including the duration of pre-training, fine-tuning, and testing a model per epoch and batch, as well as GPU memory consumption for each job. Creating an initial DS-Pre model takes the longest amount of time, while fine-tuning consumes relatively fewer resources. Once training is complete, testing can be performed much faster, processing at a speed that is, for example, 194 times faster than DS-Pre generation. It should be noted that GPU memory consumption may vary depending on different hyperparameter settings such as batch size, the number of attention/hidden layers, and the maximum length of input.

Answer to RQ4. DeepSemantic can be an efficient solution to be applicable for a downstream task that requires the semantics of binary code.

H. VISUALIZATION OF NORMALIZED INSTRUCTIONS

Figure 11 illustrates an example of how the choice of head and layer in BERT affects the model's attention patterns, generated by BERTViz [60]. This visualization shows six layers, each with eight heads. In this case, the first Head of the third Layer pays attention on tokens `[CLS]` and `[SEP]` as keys while predicting the query `add_reg8_reg4`.

VII. DISCUSSION AND LIMITATION

This section discusses several cases to consider, feasible applications for future research, and limitations of our work.

A. FUNCTION INLINING AND SPLITTING

During compilation, optimization techniques may involve with function inlining where one function is incorporated into another for better performance. This means that the labels in the binary similarity classification problem may be slightly distorted with the presence of inlining and splitting. Say, a binary compiled with `-O3` has Function A that includes Function B where the one with `-O0` has both Function A and B separately. Although the function name A stays intact but the actual content (*i.e.*, instructions) may differ significantly in the highly optimized binary. In a similar vein, a function may be split into multiple parts during compilation

B. COMPARISON GRANULARITY

Although most of functions consist of four basic blocks or 25 instructions on average (as shown in Figure 4), there are some functions with much large sizes (as a long tail). In such cases, comparison at the basic block level would be more appropriate to identify similar blocks as demonstrated in prior work [67], [69].

C. APPLICABLE DOWNSTREAM TASKS

DeepSemanticis designed to support a wide range of other downstream tasks that require semantic-aware code representation, including but not limited to a special type of vulnerability scanning, software plagiarism detection, malware behavior detection, malware family classification, and bug patch detection. It can also be used to identify a function or library in a database like IDA FLIRT [26].

D. NORMALIZATION FOR RARELY APPEARED INSTRUCTIONS

It is important to note that instructions that appear rarely during pre-training may have embeddings that are not meaningfully updated (*e.g.*, close to an initial value). Figure 3 shows that around 8.5% of the 17,221 normalized instructions in our corpus only appear once. As a result, the embedding of these instructions may not capture a meaningful context unless the number of appearance is sufficient.

E. LIMITATIONS AND FUTURE WORK

First, the current version of DeepSemanticis designed to merely handle benign binaries, and, hence, may not be directly applicable malware that often ships with various packing, obfuscation or encryption. Second, we have yet tested the performance of DeepSemanticon cross-architecture scenarios, which we leave as part of our future work. Third, while we have attempted to make our corpus diverse by including varying sizes, types (*e.g.*, executable, library), functionalities (*e.g.*, compiler, interpreter, compression, server, benchmark, AI-relevant code), and languages(*e.g.*, C/C++/Fortran) of binaries, the limited number of binaries in our corpus may impact its representativeness. Fourth, albeit rare, it is possible for two different functions to have identical instructions after normalization, which may lead to confusion for the current DeepSemantic model. Additionally, different corpora or hyperparameter settings during pre-training may affect the overall performance of DeepSemantic for a downstream task. Finally, while we have used the BERT architecture in our work, other advanced architectures like RoBERTa [39] or XLNet [66], which have demonstrated better performance in popular NLP tasks, could be explored for future work.

VIII. RELATED WORK

Penetrating the characteristics of a machine-interpretable binary code has a wide range of real-world applications including i) code clone (software plagiarism) or similarity detection [14], [15], [17], [18], [40], [43], [55], [67], [68], [69], [71], [74], ii) malware family classification [28], detection [5], [6], [35], and analysis [32], [36], [72], iii) authorship prediction [31], [37], iv) known bug discovery (code search) [7], [8], [9], [19], [38], [42], [51], [52], [57], [58], v) patching analysis [20], [29], [64], and vi) toolchain provenance [48], [56]. Most of these applications pertain to binary similarity comparison. We categorize such efforts into two approaches based on the use of machine learning techniques.

A. DETERMINISTIC APPROACHES

Binary code analysis can be performed using static and dynamic techniques. Bruschi et al. [40] introduce a means to detect code clones with the longest common subsequence of semantically equivalent basic blocks. In a similar vein, Esh [15] leverages data-flow slices of basic blocks to detect binary similarity, and later extending the idea through re-optimization [14] to improve performance. Blanket execution [18] employs dynamic equivalence testing based on seven major features. In the context of malware analysis, static features have been widely adopted for i) malware classification by extracting static features [28], ii) malware detection with structural similarities between multiple mutations [35] or control-flow graph matching [5], [6], and iii) malware authorship inference [31], [37]. Meanwhile, varying approaches have been proposed in the field of known bug discovery by leveraging i) a tree edit distance between the signature of a target basic block and that of other basic blocks [52], ii) the input/output behavior of basic blocks from intermediate representation lifting [51], and iii) dissimilar code filtering with manual features such as numeric and structural information [19]. Further, Genius [58] and FERMADYNE [9] devise another bug search engine for IoT firmware with both statistical and structural features where BinGo [7] supports both cross-architecture and cross-OS with a selective inlining technique to capture function semantics. Oftentimes, static analysis can suffer from scalability issues, such as expensive graph matching algorithms or path explosion, and may not handle structural differences (*e.g.*, optimization) well. Dynamic analysis, on the other hand, can suffer from incomplete code coverage (*e.g.*, relying on inputs) and behavior undecidability.

B. PROBABILISTIC APPROACHES

Recent advancements in machine learning techniques have received considerable attention for their applicability in binary analysis, addressing both effectiveness and efficiency. Malware analysis is one of popular applications: i) BinDNN [36] leverages deep neural networks (*e.g.*, LSTM) for function matching for malware, ii) Yueduan [72] present dynamic malware analysis with feature engineering (API calls), and iii) Neurlux [32] proposes a system that learns features automatically from a dynamic analysis report (*i.e.*, behavioral information of malware). Code similarity detection is another active domain using a probabilistic approach. Gemini [68] presents a cross-platform binary code similarity detection with a graph embedding network and Siamese network [4]. Lately, varying efforts to deduce underlying semantics of a binary code have been made with *deep learning*. InnerEye [74] aims to detect code similarity across different architectures, borrowing the concept of neural machine translation (NMT) from an NLP domain. It generates vocabulary embeddings with a Word2vec Skipgram [44] model with a coarse-grained normalization process. Similarly, MIRROR [71] presents a basic block embedding means across different ISAs that utilizes an NMT model to establish connections between the two ISAs. Asm2vec [17] applies a PV-DM model for code clone search, demonstrating that code representation can be robust over compiler optimization and even code obfuscation. SAFE [43] generates function embedding based on a self-attentive neural network. DeepBinDiff [67] performs a binary similarity task at the basic block granularity with token embeddings for semantic information, feature vectors, and the TADW algorithm for program-wide contextual information. For comparison, our experiment includes the two state-of-the-art approaches, SAFE and DeepBinDiff. Meanwhile, Yu et al. [55] investigate the graph embedding network that extracts relevant features automatically, resulting in similar performance compared to the architecture without using any structural information. Note that DeepSemantic use limited call graph information (*e.g.*, `libc` call).

C. COMPARISON WITH THE PREVIOUS STUDY

One of the closest work to ours in terms of adopting the BERT architecture is *Order matters* [69]. The main difference is that *Order matters* focuses on seeking identical binaries by representing vectors for different binaries (*i.e.*, identical source but dissimilar binaries due to different platform and optimization) as close as possible. In contrast, our proposed model performs contextual similarity detection between binary functions. For evaluation, *Order matters* employs rank-aware metrics such as Top1, MRR (Mean Reciprocal Rank), and NDCG (Normalized Discounted Cumulative Gain), and reports an accuracy of 74%. In comparison, our evaluation employs a classification metric. However, we were unable to compare our results with *Order matters* due to the unavailability of their source code. While we could not replicate their work, it is worth noting that constructing an implementation based solely on conceptual information [69] is not possible without implementation details such as model hyperparameters.

IX. CONCLUSION

In this paper, we propose a well-balanced instruction normalization technique that aims to convey as much information

TABLE 8. Top 144 normalized instructions and their frequencies in our dataset. The last 14% of total instructions has rarely seen with a long tail of the distribution (Figure 3). The group field indicates (C)all, (J)ump and (M)ove instructions.

Rank	Normalized Instruction	Ratio	Cumulative	Group	Rank	Normalized Instruction	Ratio	Cumulative	Group
1	mov_reg8_reg8	8.243%	8.24%	M	73	movzx_reg4_wordptr[reg8]	0.163%	76.04%	M
2	call_innerfunc	5.735%	13.98%	C	74	sub_reg4_immval	0.163%	76.20%	
3	mov_reg8_qwordptr[bp8-disp]	4.164%	18.14%	M	75	mov_dwordptr[bp8-disp]_immval	0.157%	76.36%	M
4	je_jmpdst	3.925%	22.07%	J	76	mov_qwordptr[ip8+disp]_reg8	0.156%	76.51%	M
5	jmp_jmpdst	3.859%	25.93%	J	77	mov_qwordptr[sp8+disp]_immval	0.154%	76.67%	M
6	mov_reg4_immval	3.486%	29.41%	M	78	cmp_qwordptr[bp8-disp]_immval	0.153%	76.82%	
7	jne_jmpdst	2.613%	32.03%	J	79	mov_qwordptr[reg8+8]_reg8	0.151%	76.97%	M
8	mov_reg8_qwordptr[reg8+disp]	2.267%	34.29%	M	80	mov_qwordptr[sp8+8]_reg8	0.151%	77.12%	M
9	pop_reg8	1.936%	36.23%		81	cmp_dwordptr[bp8-disp]_immval	0.151%	77.27%	
10	mov_reg4_reg4	1.860%	38.09%	M	82	mov_reg4_bp4	0.150%	77.42%	M
11	push_reg8	1.844%	39.93%		83	cdqe	0.146%	77.57%	
12	mov_qwordptr[bp8-disp]_reg8	1.839%	41.77%	M	84	test_reg1_immval	0.141%	77.71%	
13	xor_reg4_reg4	1.725%	43.50%		85	jg_jmpdst	0.140%	77.85%	J
14	ret	1.477%	44.97%	C	86	mov_reg8_sp8	0.139%	77.99%	M
15	test_reg8_reg8	1.387%	46.36%		87	mov_reg4_dispbss	0.137%	78.13%	M
16	mov_reg8_qwordptr[reg8]	1.370%	47.73%	M	88	shr_reg4_immval	0.136%	78.26%	
17	mov_reg8_qwordptr[sp8+disp]	1.306%	49.04%	M	89	mov_reg8_qwordptr[reg8+reg8*8]	0.134%	78.40%	M
18	cmp_reg4_immval	1.111%	50.15%		90	mov_reg8_qwordptr[sp8]	0.134%	78.53%	M
19	add_reg8_immval	1.107%	51.25%		91	call_reg8	0.132%	78.66%	C
20	call_externfunc	1.061%	52.31%	C	92	mulsd_regxmm_regxmm	0.128%	78.79%	
21	mov_reg4_dispstr	1.046%	53.36%	M	93	movabs_reg8_immval	0.127%	78.92%	M
22	test_reg4_reg4	0.936%	54.30%		94	mov_dwordptr[reg8+disp]_immval	0.126%	79.04%	M
23	push_bp8	0.891%	55.19%		95	shr_reg8_immval	0.126%	79.17%	
24	mov_reg4_dwordptr[bp8-disp]	0.872%	56.06%	M	96	movzx_reg4_byteptr[reg8]	0.126%	79.30%	M
25	sub_sp8_immval	0.812%	56.87%		97	mov_bp4_immval	0.124%	79.42%	M
26	mov_qwordptr[sp8+disp]_reg8	0.808%	57.68%	M	98	sar_reg8_immval	0.123%	79.54%	
27	mov_reg8_qwordptr[bp8-8]	0.801%	58.48%	M	99	cmp_reg2_immval	0.117%	79.66%	
28	add_sp8_immval	0.786%	59.26%		100	jl_jmpdst	0.116%	79.78%	J
29	pop_bp8	0.757%	60.02%		101	lea_reg8_[reg8+reg8]	0.115%	79.89%	
30	mov_reg8_qwordptr[ip8+disp]	0.690%	60.71%	M	102	sub_reg4_reg4	0.114%	80.01%	
31	lea_reg8_[bp8-disp]	0.680%	61.39%		103	mov_reg1_immval	0.113%	80.12%	M
32	mov_reg8_qwordptr[reg8+8]	0.680%	62.07%	M	104	mov_reg8_qwordptr[disp]	0.112%	80.23%	M
33	lea_reg8_[sp8+disp]	0.651%	62.72%		105	cmp_qwordptr[reg8+disp]_immval	0.112%	80.34%	
34	mov_reg8_bp8	0.622%	63.35%	M	106	cmp_dwordptr[ip8+disp]_immval	0.111%	80.45%	
35	cmp_reg8_reg8	0.619%	63.96%		107	vmulsd_regxmm_regxmm_regxmm	0.109%	80.56%	
36	mov_reg4_dwordptr[reg8+disp]	0.610%	64.57%	M	108	lea_reg8_[reg8+reg8*8]	0.107%	80.67%	
37	mov_dwordptr[bp8-disp]_reg4	0.584%	65.16%	M	109	mov_dwordptr[sp8+disp]_immval	0.106%	80.78%	M
38	mov_bp8_sp8	0.569%	65.73%	M	110	movabs_reg8_dispstr	0.106%	80.88%	M
39	add_reg8_reg8	0.550%	66.28%		111	sete_reg1	0.105%	80.99%	
40	mov_qwordptr[reg8+disp]_reg8	0.512%	66.79%	M	112	jge_jmpdst	0.103%	81.09%	J
41	mov_reg4_dispdata	0.470%	67.26%	M	113	cmp_byteptr[reg8+disp]_immval	0.101%	81.19%	
42	and_reg4_immval	0.466%	67.73%		114	js_jmpdst	0.100%	81.29%	J
43	mov_qwordptr[bp8-8]_reg8	0.459%	68.19%	M	115	mov_dwordptr[reg8]_reg4	0.100%	81.39%	M
44	test_reg1_reg1	0.426%	68.61%		116	sub_reg8_immval	0.099%	81.49%	
45	shl_reg8_immval	0.422%	69.03%		117	shl_reg4_immval	0.099%	81.59%	
46	add_reg4_immval	0.397%	69.43%		118	test_byteptr[reg8+disp]_immval	0.096%	81.68%	
47	lea_reg8_[reg8+disp]	0.349%	69.78%		119	movapd_regxmm_regxmm	0.096%	81.78%	M
48	movsxd_reg8_reg4	0.332%	70.11%	M	120	mov_qwordptr[sp8]_reg8	0.093%	81.87%	M
49	mov_bp8_reg8	0.325%	70.44%	M	121	mov_byteptr[bp8-disp]_reg1	0.093%	81.97%	M
50	mov_reg4_dwordptr[sp8+disp]	0.322%	70.76%	M	122	mov_reg8_qwordptrfs:[disp]	0.093%	82.06%	M
51	mov_qwordptr[reg8]_reg8	0.299%	71.06%	M	123	xor_reg8_qwordptrfs:[disp]	0.093%	82.15%	
52	mov_reg4_dwordptr[reg8]	0.284%	71.34%	M	124	cmp_dwordptr[reg8+disp]_immval	0.093%	82.24%	
53	call_qwordptr[reg8+disp]	0.280%	71.62%	C	125	push_immval	0.092%	82.34%	
54	ja_jmpdst	0.278%	71.90%	J	126	movsxd_reg8_dwordptr[bp8-disp]	0.091%	82.43%	M
55	mov_reg4_dwordptr[ip8+disp]	0.262%	72.16%	M	127	setne_reg1	0.091%	82.52%	
56	cmp_reg4_reg4	0.245%	72.41%		128	mov_reg4_dwordptr[reg8+8]	0.091%	82.61%	M
57	jae_jmpdst	0.240%	72.65%	J	129	test_bp8_bp8	0.091%	82.70%	
58	jle_jmpdst	0.239%	72.89%	J	130	lea_reg8_[reg8+reg8*2]	0.090%	82.79%	
59	sub_reg8_reg8	0.238%	73.12%		131	lea_reg8_[reg8+1]	0.088%	82.88%	
60	cmp_reg8_immval	0.236%	73.36%		132	movzx_reg4_reg2	0.086%	82.96%	M
61	jbe_jmpdst	0.232%	73.59%	J	133	lea_reg4_[reg8+1]	0.086%	83.05%	
62	mov_dwordptr[sp8+disp]_reg4	0.230%	73.82%	M	134	mov_qwordptr[reg8]_dispdata	0.086%	83.13%	M
63	mov_qwordptr[reg8+disp]_immval	0.227%	74.05%	M	135	mov_bp4_reg4	0.084%	83.22%	M
64	mov_reg8_qwordptr[bp8+disp]	0.225%	74.27%	M	136	vmovsd_regxmm_qwordptr[sp8+disp]	0.084%	83.30%	
65	mov_reg8_qwordptr[sp8+8]	0.224%	74.50%	M	137	addsd_regxmm_regxmm	0.083%	83.38%	
66	movzx_reg4_reg1	0.216%	74.71%	M	138	mov_qwordptr[bp8-disp]_immval	0.082%	83.47%	M
67	add_reg4_reg4	0.211%	74.92%		139	vmovsd_qwordptr[sp8+disp]_regxmm	0.081%	83.55%	
68	jb_jmpdst	0.211%	75.14%	J	140	and_reg1_immval	0.080%	83.63%	
69	mov_dwordptr[reg8+disp]_reg4	0.195%	75.33%	M	141	mov_reg8_qwordptr[bp8]	0.080%	83.71%	M
70	cmp_reg1_immval	0.193%	75.52%		142	pxor_regxmm_regxmm	0.080%	83.79%	
71	leave	0.186%	75.71%	C	143	lea_reg8_[reg8+8]	0.079%	83.87%	
72	movzx_reg4_byteptr[reg8+disp]	0.167%	75.88%	M	144	mov_byteptr[reg8+disp]_immval	0.078%	83.94%	M

as possible for a deep learning architecture. To demonstrate the effectiveness of normalization approach, we introduce DeepSemantic that leverages the BERT architecture, to build a pre-trained model for generic code representation, and a downstream model that requires the inference of code semantics. Our experimental results, specifically with the DS-BinSim downstream task, show that DeepSemanticoutperforms existing binary similarity comparison tools.

APPENDIX

See Table 8.

ACKNOWLEDGMENT

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the sponsor's views.

REFERENCES

[1] J. Alammar. (2018). *The Illustrated BERT, ELMo, and Co. (How NLP Cracked Transfer Learning)*. [Online]. Available: https://jalammar.github.io/illustrated-bert/

[2] Angr. (2016). *Python Framework for Analyzing Binaries*. [Online]. Available: https://angr.io/

[3] D. Bahdanau, K. H. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.

[4] J. Bromley, I. M. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a 'Siamese' time delay neural network," in *Proc. 6th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 1993, pp. 1–8.

[5] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Proc. 3rd Int. Conf., (DIMVA)*, Jul. 2006, pp. 129–143.

[6] S. Cesare, Y. Xiang, and W. Zhou, "Control flow-based malware variant detection," *IEEE Trans. Depend. Secure Comput.*, vol. 11, no. 4, pp. 307–317, Jul./Aug. 2014.

[7] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 678–689.

[8] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 896–899.

[9] D. D. Chen, M. Egele, M. Woo, and D. Brumley, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. NDSS*, Feb. 2016, p. 1.

[10] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1–15.

[11] Y.-H. Choi, P. Liu, Z. Shang, H. Wang, Z. Wang, L. Zhang, J. Zhou, and Q. Zou, "Using deep learning to solve computer security challenges: A survey," *Cybersecurity*, vol. 3, no. 1, pp. 1–13, Dec. 2020.

[12] A. M. Dai and Q. V. Le, "Semi-supervised sequence learning," 2015, *arXiv:1511.01432*.

[13] I. U. Haq and J. Caballero, "A survey of binary code similarity," 2019, *arXiv:1909.11424*.

[14] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 79–94, Sep. 2017.

[15] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 266–280, Aug. 2016.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol.*, vol. 1, Jun. 2019, pp. 4171–4186.

[17] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.

[18] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.*, 2014, pp. 303–317.

[19] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 58–79.

[20] H. Flake, "Structural comparison of executable objects," in *Proc. GI SIG SIDAR Workshop, (DIMVA). Gesellschaft Für Informatik eV*, 2004, pp. 1–13.

[21] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–36, Jul. 2018.

[22] Google. (2020). *End-to-End Open Source Machine Learning Platform*. [Online]. Available: https://tensorflow.org

[23] Google. (2020). *Release of BERT Models*. [Online]. Available: https://github.com/google-research/bert

[24] Hex-Rays. (2005). *IDA Pro Disassembler*. [Online]. Available: https://www.hex-rays.com/products/ida/

[25] Hex-Rays. (2019). *IDAPython Documentation*. [Online]. Available: https://www.hex-rays.com/products/ida/support/idapython_docs/

[26] Hex-Rays. (2020). *IDA Pro Fast Library Identification and Recognition Technology*. [Online]. Available: https://www.hex-rays.com/products/ida/tech/flirt/

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin, "MutantX-S: Scalable malware clustering based on static features," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2013, pp. 187–198.

[29] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Cross-architecture binary semantics understanding via similar code comparison," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, Mar. 2016, pp. 57–67.

[30] huanghonggit. (2019). *BERT MLM with Pytorch*. [Online]. Available: https://github.com/huanghonggit/Mask-Language-Model

[31] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. USENIX Secur. Symp.*, 2013, pp. 81–96.

[32] C. Jindal, C. Salls, H. Aghakhani, K. Long, C. Kruegel, and G. Vigna, "Neurlux: Dynamic malware analysis without feature engineering," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, Dec. 2019, pp. 444–455.

[33] A. Karpathy. (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. [Online]. Available: http://karpathy.github.io/2015/05/21/rnn-effectiveness

[34] T. Kim, Y. R. Lee, B. Kang, and E. G. Im, "Binary executable file similarity calculation using function matching," *J. Supercomput.*, vol. 75, no. 2, pp. 607–622, Feb. 2019.

[35] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Proc. RAID*, 2005, pp. 207–226.

[36] N. Lageman, E. D. Kilmer, R. J. Walls, and P. D. McDaniel, "BINDNN: Resilient function matching using deep learning," in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2016, pp. 517–537.

[37] M. Lindorfer, A. Di Federico, F. Maggi, P. M. Comparetti, and S. Zanero, "Lines of malicious code: Insights into the malicious software industry," in *Proc. 28th Annu. Comput. Secur. Appl. Conf.*, Dec. 2012, pp. 349–358.

[38] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αDiff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, Sep. 2018, pp. 667–678.

[39] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[40] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2014, pp. 389–400.

[41] L. Massarelli. (2019). *Self Attentive Function Embedding Tool*. [Online]. Available: https://github.com/gadiluna/SAFE

[42] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proc. Workshop Binary Anal. Res.*, 2019, pp. 1–11.

[43] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Safe: Self-attentive function embeddings for binary similarity," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2019, pp. 309–329.

[44] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2013, pp. 3111–3119.

[45] Nginx. (2020). *High Performance Load-Balancer and Web Server*. [Online]. Available: https://nginx.com

[46] (NSA), N.S.A. (2019). *Software Reverse Engineering (SRE) Suite of Tools*. [Online]. Available: https://ghidra-sre.org/

[47] OpenSSL. (2020). *Cryptography and SSL/TLS Toolkit*. [Online]. Available: https://www.openssl.org

[48] Y. Otsubo, A. Otsuka, M. Mimura, T. Sakaki, and H. Ukegawa, "O-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment," in *Proc. Workshop Binary Anal. Res.*, 2020, pp. 1–12.

[49] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2021, pp. 833–851.

[50] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Hum. Lang. Technol., (Long Papers)*, vol. 1, 2018, pp. 2227–2237.
[51] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," *IT Inf. Technol.*, vol. 59, no. 2, pp. 83–91, Apr. 2017.
[52] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. 30th Annu. Comput. Secur. Appl. Conf.*, Dec. 2014, pp. 406–415.
[53] PyTorch. (2019). *Open Source Machine Learning Framework*, [Online]. Available: https://pytorch.org/
[54] Radare2. (2019). *Libre and Portable Reverse Engineering Framework*. [Online]. Available: https://rada.re/n/
[55] K. Redmond, L. Luo, and Q. Zeng, "A cross-architecture instruction embedding model for natural language processing-inspired binary code analysis," in *Proc. Workshop Binary Anal. Res.*, 2019, pp. 1–8.
[56] N. Rosenblum, B. P. Miller, and X. Zhu, "Recovering the toolchain provenance of binary code," in *Proc. Int. Symp. Softw. Test. Anal.*, Jul. 2011, pp. 100–110.
[57] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna, "Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic device," in *Proc. Detection Intrusions Malware, Vulnerability Assessment, 15th Int. Conf., (DIMVA)*, Saclay, France, Jun. 2018, pp. 114–138.
[58] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 480–491.
[59] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. 31th Conf. Neural Inf. Process. Syst. (NeurIPS)*, 2017, pp. 1–15.
[60] J. Vig, "A multiscale visualization of attention in the transformer model," 2019, *arXiv:1906.05714*.
[61] Vsftpd. (2020). *Probably the Most Secure and Fastest FTP Server*. [Online]. Available: https://security.appspot.com/vsftpd.html
[62] Wikipedia. (2020). *Vanishing Gradient Problem*. [Online]. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem
[63] T. Wolf, "HuggingFace's Transformers: State-of-the-art natural language processing," 2019, *arXiv:1910.03771*.
[64] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security patch analysis for binaries towards understanding the pain and pills," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 462–472.
[65] H. Xue, S. Sun, G. Venkataramani, and T. Lan, "Machine learning-based analysis of program binaries: A comprehensive study," *IEEE Access*, vol. 7, pp. 65889–65912, 2019.
[66] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized autoregressive pretraining for language understanding," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–18.
[67] Y. Duan, X. Li, J. Wang, and H. Yin, "DeepBinDiff: Learning program-wide code representations for binary diffing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2020, pp. 1–16.
[68] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.
[69] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: Semantic-aware neural networks for binary code similarity detection," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 1145–1152.
[70] Yueduan. (2020). *Fine-Grained Binary Diffing Tool for x86 Binaries*. [Online]. Available: https://github.com/yueduan/DeepBinDiff
[71] X. Zhang, W. Sun, J. Pang, F. Liu, and Z. Ma, "Similarity metric method for binary basic blocks of cross-instruction set architecture," in *Proc. Workshop Binary Anal. Res.*, 2020, pp. 1–12.
[72] Z. Zhang, P. Qi, and W. Wang, "Dynamic malware analysis with feature engineering and feature learning," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 1210–1247.
[73] P. S. Florence and G. K. Zipf, "Human behaviour and the principle of least effort," *Econ. J.*, vol. 60, no. 240, p. 808, Dec. 1950.
[74] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2019, pp. 1–15.

HYUNGJOON KOO received the M.Sc. degree in information security from Korea University, in 2010, and the Ph.D. degree in computer science from Stony Brook University, in 2019. He is currently an Assistant Professor with the Department of Computer Science and Engineering, College of Computing, Sungkyunkwan University (SKKU). Before joining SKKU, he was a Postdoctoral Researcher with the SSLab, Georgia Institute of Technology. He is leading the Security with AI (SecAI) Laboratory. His research interests include software security, network security, binary analysis and protection, and security leveraging artificial intelligence.

SOYEON PARK received the bachelor's degree from the Pohang University of Science and Technology (POSTECH). She is currently pursuing the Ph.D. degree with the School of Cybersecurity and Privacy (SCP) and the School of Computer Science (SCS), Georgia Institute of Technology, under the supervision of Prof. Taesoo Kim. Her research has been supported in part by the Georgia Tech IISP Cybersecurity Fellowship. Her papers have been published in several conferences, including ACSAC, ACM CCS, IEEE S&P, and USENIX ATC. Her research interests include software security, with a focus on automatic vulnerability detection in a real-world application, hardware-assisted memory hardening, and binary analysis with machine learning.

DAEJIN CHOI received the B.S. degree in computer science and engineering from Korea University, in 2012, and the Ph.D. degree in computer science and engineering from Seoul National University, in 2019. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Incheon National University (INU). Before joining INU, he was a Research Scientist with the Georgia Institute of Technology. His research interests include applied machine learning, human-centered AI, social computing, and data mining.

TAESOO KIM received the B.S. degree from KAIST, in 2009, and the S.M. and Ph.D. degrees from MIT, in 2011 and 2014, respectively. He is currently a Professor with the School of Cybersecurity and Privacy (SCP) and the School of Computer Science (SCS), Georgia Institute of Technology (Georgia Tech). He is also the Director of the Georgia Tech Systems Software and Security Center (GTS3). Starting from his sabbatical year, he works as the Vice President of Samsung Research, leading the development of a Rust-based OS for a secure element. He has published more than 100 papers in top systems and security conferences. He was a recipient of several awards, including the NSF CAREER, in 2018, the Internet Defense Prize, in 2015, and several best paper awards, including the USENIX Security 2018 and the EuroSys 2017.