sem2vec: Semantics-Aware Assembly Tracelet Embedding

HUAIJIN WANG, Hong Kong University of Science and Technology, China
PINGCHUAN MA, Hong Kong University of Science and Technology, China
SHUAI WANG*, Hong Kong University of Science and Technology, China
QIYI TANG, Keen Security Lab, Tencent, China
SEN NIE, Keen Security Lab, Tencent, China
SHI WU, Keen Security Lab, Tencent, China

Binary code similarity is the foundation of many security and software engineering applications. Recent works leverage deep neural networks (DNN) to learn a numeric vector representation (namely *embeddings*) of assembly functions, enabling similarity analysis in the numeric space. However, existing DNN-based techniques capture syntactic-, control flow-, or data flow-level information of assembly code, which is too coarse-grained to represent program functionality. These methods can suffer from low robustness to challenging settings such as compiler optimizations and obfuscations.

We present sem2vec, a binary code embedding framework that learns from *semantics*. Given the control-flow graph (CFG) of an assembly function, we divide it into *tracelets*, denoting continuous and short execution traces that are reachable from the function entry point. We use symbolic execution to extract symbolic constraints and other auxiliary information on each tracelet. We then train masked language models to compute embeddings of symbolic execution outputs. Last, we use graph neural networks, to aggregate tracelet embeddings into the CFG-level embedding for a function. Our evaluation shows that sem2vec extracts high-quality embedding and is robust against different compilers, optimizations, architectures, and popular obfuscation methods including virtualization obfuscation. We further augment a vulnerability search application with embeddings computed by sem2vec and demonstrate a significant improvement in vulnerability search accuracy.

CCS Concepts: • **Security and privacy → Software security engineering**.

Additional Key Words and Phrases: Symbolic Execution, Embedding, Graph Neural Network, Binary Code Similarity

1 INTRODUCTION

Binary code matching is the core building block of many important software engineering and security applications. For example, malware analysis compares suspicious code to known malware families to determine whether it is malicious [33, 45]. Similarly, given a known software vulnerability, security analysts must frequently decide its presence in real-world software in executable format [20]. Binary code matching also helps discover code clones and algorithm plagiarism [60, 76, 98].

*Corresponding author

Authors' addresses: Huaijin Wang, hwangdz@cse.ust.hk, Hong Kong University of Science and Technology, Clear Water Bay, Sai Kung, Hong Kong, China; Pingchuan Ma, pmaab@cse.ust.hk, Hong Kong University of Science and Technology, Clear Water Bay, Sai Kung, Hong Kong, China; Shuai Wang, shuaiw@cse.ust.hk, Hong Kong University of Science and Technology, Clear Water Bay, Sai Kung, Hong Kong, China; Qiyi Tang, dodgetang@tencent.com, Keen Security Lab, Tencent, Tianlin Road, Xuhu, Shanghai, China; Sen Nie, snie@tencent.com, Keen Security Lab, Tencent, Tianlin Road, Xuhu, Shanghai, China; Shi Wu, shiwu@tencent.com, Keen Security Lab, Tencent, Tianlin Road, Xuhu, Shanghai, China.

1049-331X/2022/10-ART $15.00
https://doi.org/10.1145/3569933

The AI community has made major advancements in deep learning and representation learning [29, 70, 88]. For example, to compare two natural language sentences, instead of explicitly specifying features in sentences for comparison, representation learning trains a model to learn the most representative aspects from sentences for comparison gradually. Typically, representation learning computes a numeric vector, termed *embedding*, for each input, and the cosine distance of two embeddings determines the similarity of two inputs. Representation learning has been widely used to understand natural language text, images, and graphs. Recent research has shown the feasibility of conducting representation learning over binary code [13, 15, 19, 30, 55, 58, 92, 99]. This was often done by converting basic blocks into numeric vectors and then using graph neural networks to compute embeddings over program control structures.

Unlike high-dimensional data, such as an image, whose visual appearance are usually sufficient for similarity comparison, the software is *more subtle*. Deep learning models may not generate quality embeddings that faithfully reflect a program's underlying functionality [85, 87]. Two programs of different functionalities may look "similar" from the token or graph perspective. Conversely, two programs may look different due to code obfuscation or compiler optimizations, although they have identical functionality. These intrinsic obstacles make binary code matching very difficult, as is demonstrated in Sec. 3.

Key Idea. We present sem2vec, a tool to compute robust binary code embeddings. sem2vec uses a hybrid approach by leveraging symbolic execution (SE) and deep learning techniques. In general, SE is good at extracting *precise* semantics-level signatures regarding input/output constraints and path constraints. However, SE usually suffers from low scalability owing to path explosion and the large number of symbolic states. Meanwhile, existing deep learning techniques, particularly masked language models (MLMs) and graph neural networks (GNNs) (or "graph embedding network" in Gemini's own terminology [92]), are good at learning a *scalable* view of code representations, but generally fail to understand subtle semantics in a precise manner. sem2vec, for the first time, explores a rational and novel combination of SE and MLM/GNN, which benefits from a *synergistic effect* and achieves a highly promising learning quality. sem2vec divides a function-level CFG into continuous and short traces (i.e.,*tracelets*) and performs SE on the tracelets to compute precise semantics signatures (i.e., symbolic constraints). We then train MLMs to convert each symbolic constraint into an embedding vector and use GNNs to aggregate tracelet embeddings into a joint embedding of the entire CFG.

We bridge tracelet embeddings generated by sem2vec with two CFG embedding pipelines, BinaryAI [93] and Gemini [92]. To evaluate sem2vec, we form a large-scale dataset including Linux coreutils [4], binutils [3], findutils [6], diffutils [5], OpenSSL [10], libtomcrypt [9], libgmp [7], zlib [12], and rapidjson [2], with a total of **116,941,424** pairs of assembly functions for comparison and **713,209,256** assembly instructions. We also leverage a vulnerability database used in prior works [28, 30] to assess how sem2vec boosts vulnerability search in real-world executables. First, we compile programs on the 64-bit x86 platform, using two compilers (gcc and clang) and three optimization levels. We also adopt a common software obfuscator (llvm-Obfuscator [50]) with four obfuscation schemes. Executables compiled from the same source code but with different settings (e.g., obfuscations) are substantially changed; nevertheless, sem2vec generates binary code embedding of high robustness against changes introduced by obfuscation and optimizations. sem2vec is scalable to complete control-flow graph-level symbolic execution with about two CPU minutes for an assembly function. Evaluation results show that sem2vec achieves the highest top-1 accuracy and outperforms the state-of-the-art tools on 37 out of 50 comparison settings including challenging cross-compiler, cross-optimization and obfuscation settings. For the rest 13 settings, sem2vec exhibits the second-highest top-1 scores of 12, and all top-1 scores are close to the best tool, BinaryAI [93]. sem2vec outperforms another recent work, PalmTree [54], in nearly all settings. sem2vec also effectively augments vulnerability searching by matching CVEs (e.g., Heartbleed) at top-1 for 11 out of 12 cases, outperforming all evaluated tools. Additionally, to illustrate the generalization of sem2vec toward different graph neural network backends, different architectures, and more heavyweight obfuscation schemes, we setup evaluations using different graph neural networks (e.g., GTN [96]), virtualization-based obfuscation (provided by `Tigress` [17]), and cross-architecture comparisons (i.e., binaries compiled on x86 vs. binaries compiled on aarch64). We show that `sem2vec` achieves consistently encouraging results across all the challenging setups. In summary, we make the following contributions:

- We propose a new perspective to computing binary code embedding by learning from program semantics. Quality embeddings that are resilient to various challenging obfuscation, compilation and optimization settings can be produced.
- We present a practical tool, `sem2vec`, that performs hybrid analysis by incorporating scalable tracelet-level SE, MLMs, and GNNs to achieve a *synergistic effect*. Tracelet-level SE provides precise semantics signatures at a low cost, and neural models learn more holistic structure-level representations.
- Our evaluation shows that `sem2vec` generates high-quality embeddings and is robust in cross-compiler, optimization, and obfuscation settings. Vulnerability search engines are augmented by `sem2vec` with much higher accuracy.

Artifact. We provide the artifact of this research at [11].

2 PRELIMINARIES

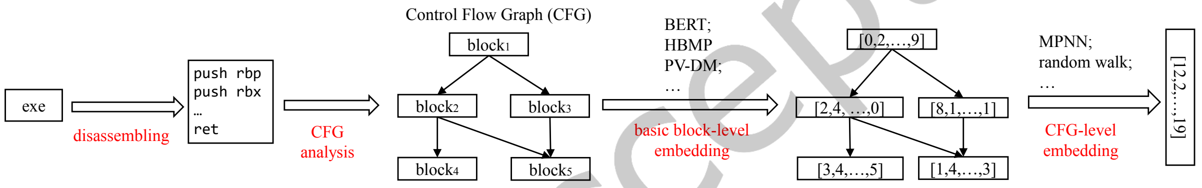

Fig. 1. Binary code embedding overview.

This section introduces code embedding techniques. Although our example employs binary code, the general procedure can be used to construct source code embeddings. Nearly all embedding models compute a standalone embedding for an assembly function [13, 15, 19, 30, 55, 58, 92, 99]. For simplicity, the rest of this section assumes the input executable has only one function.

As seen in Fig. 1, binary code embedding usually comprises three steps. The pre-processing module disassembles the input executable into assembly instructions. The CFG is also recovered over each assembly function and supplied into the following modules. Then, the basic block embedding module and the graph embedding module collaborate to embed the CFG. The final output is a graph embedding that represents each assembly function numerically. A well-performing embedding framework can locate and include rich information; two similar assembly functions will have embedding vectors of a short cosine distance. We now explain each step.

Basic Block-Level Embedding. This step often treats machine instructions within each block as a natural language paragraph, where each instruction i deems a sentence. Instruction opcodes and operands are considered as words. MLMs, like PV-DM [53], are often used [30]. In general, an embedding model M_b, which extracts tokens from an instruction i and maps i into an embedding vector v, needs to be trained. To do so, we can iteratively mask one instruction i_t, and map other instructions $i \in I$ surrounding i_t into vectors $v \in V$. $v \in V$ is aggregated to predict i_t. The back-propagated prediction error is used to update M_b until saturation. Instead of training M_b from scratch, recent studies fine-tune pre-trained BERT-like [29] language models [93, 94]. `sem2vec`'s tracelet-level embedding, as introduced in Sec. 4.2, is based on pre-trained RoBERTa [57]; however, by feeding the model with symbolic constraints instead of tokens, we formulate different pre-training tasks.

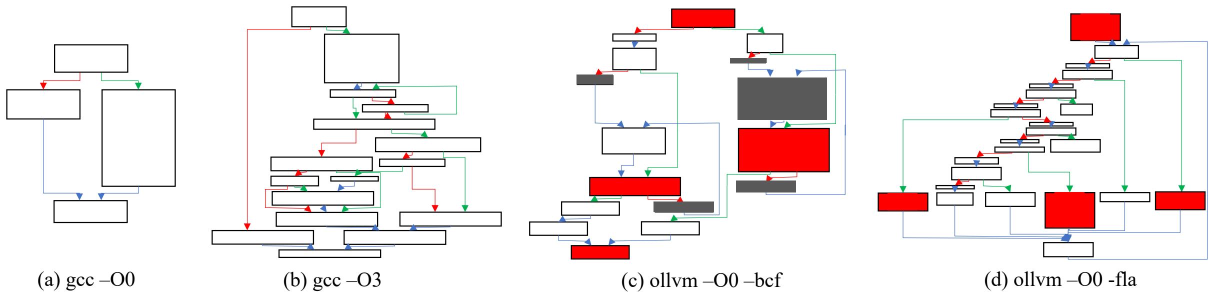

Fig. 2. Motivating example. We compare assembly functions compiled from the same source code in Linux `coreutils` [4] `chroot`. Function `_usage_chroot` is compiled using gcc `-O0`, gcc `-O3`, LLVM obfuscator `ollvm` with the control flow flattening `-fla` option, and `ollvm` with the bogus control flow `-bcf` option. For latter two cases, we mark the blocks containing user-written code in red . the rest blocks are generated by `ollvm` which make the CFG much complex.

CFG-Level Embedding. In addition to extracting basic block-level embeddings, the CFG must be recast numerically. Many graph neural networks (GNNs) models have shown promise on graph classification tasks. A well-trained GNN model has high expressiveness comparable to that of the Weisfeiler–Lehman graph isomorphism test [91]; thus, GNN models are capable of supporting tasks like software matching. Most GNNs belong to the family of message passing neural networks (MPNNs) [38]. A message-passing phase and a readout phase are both parts of the MPNN standard.

The message-passing phase updates the hidden states at each basic block based on its neighboring nodes. In our case, the "hidden states" are computed embeddings. Typically, the message-passing phase can take several iterations, and each node's embedding is gradually updated until it reaches a fixed point or a pre-defined threshold. The readout phase (also termed the aggregation phase) then computes a whole graph embedding using a pre-defined readout function, yielding the final embedding result. To augment CFG embedding, recent work has also used random walks [30]. `sem2vec` adopts GGNN [56] to aggregate embeddings of each tracelet rather than each basic block (see Sec. 4.3 for details).

3 RESEARCH MOTIVATION

Research Challenge. Existing works [30, 61, 93] have provided a solid foundation for computing high-quality binary code embeddings and enabled downstream security and code analysis tasks. Sec. 2 has shown that many works learn from program syntactic-level features, which, although easily approachable, are not *robust* to changes in binary code syntax owing to compilation, obfuscation, and optimizations.

Fig. 2 presents an example in which we compile a piece of source code into four assembly functions using various compilation/obfuscation settings. Without optimization (Fig. 2(a)), a simple CFG is formed. While applying full optimizations (Fig. 2(b)) yields a significantly more sophisticated CFG. When bogus control flow obfuscation (also termed "opaque predicate" in [52]) is applied, an opaque predicate inserts a tautology path condition that is hard to analyze; however, this condition is always evaluated in one direction at runtime (Fig. 2(c)). Extra "garbage code" (marked in grey) can be added in the unreachable branch, thus changing the visual appearance of the code. Control flow flattening (Fig. 2(d)) turns the CFG into a "flattened" structure with blocks containing user-written code organized into "case statements" (see the three red blocks near the bottom) of a gigantic "switch" statement. Note that the *functionality* of the flattened CFG is identical to the non-obfuscated version, but it differs dramatically from the CFG built with `clang -O0`.

Obfuscation and optimization often introduce challenges to existing DNN-based binary code embedding tools, given that they extract syntactic- or graph-level information, which does not necessarily reflect the real functionality. The trained embedding models may be vulnerable to assembly codes with similar functionality but differing appearance. We show that modern DNN-based embedding tools, including BinaryAI [93], ASM2VEC [30], and SAFE [61], compute embeddings of these assembly functions with a large cosine distance, i.e., they treat these four assembly functions as highly dissimilar.

Generating Robust Code Embeddings from Symbolic Constraints. In addition to the prosperous development of using DNN-based code embedding for binary similarity analysis, another relevant line of research is to perform rigorous binary equivalence checking, using program semantics constraints, often represented as input-output symbolic constraints, generated by symbolic execution (SE) [24, 26, 37, 49, 59, 63]. Overall, with a pair of symbolic constraints denoting the input-output relations of binary code (e.g., an assembly function), symbolic execution employs constraint solving techniques to prove the equivalence of their semantics. Given that the extracted symbolic constraints are indicators of code functionality rather than syntactic forms, the analysis results are resilient to challenging settings such as compiler optimizations, cross architecture settings, and even code obfuscation, since these settings change code syntactic form, but retain the original program semantics.

Overall, this work explores a unique and novel combination to use symbolic constraints, the output of performing symbolic execution toward binary code, as the input of follow-up binary code embedding learning. We anticipate that when using symbolic execution to extract precise and robust semantics representation from binary code, the learned code embedding will manifest much better robustness toward challenges like cross optimization, cross architecture, and obfuscation-involved binary code comparison. Indeed, we find that semantics extracted by sem2vec's symbolic execution module (see details in the next section) can construct embeddings of these code samples with much higher similarity from the semantics perspective. Overall, given the embeddings generated by sem2vec, three assembly functions in Fig. 2(b), Fig. 2(c), Fig. 2(d) appear in top-1 matchings of the assembly function in Fig. 2(a), as will be shown in Sec. 6.

4 DESIGN

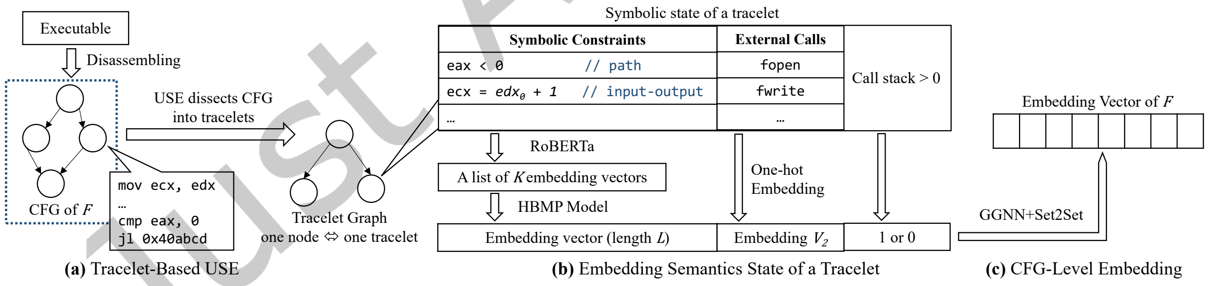

Fig. 3. Workflow of sem2vec. Consistent with most relevant works, we generate an embedding vector for each assembly function. To ease the presentation, we assume the input executable has one function.

We now introduce sem2vec, a binary code embedding framework that uses both semantic signatures extracted by SE and holistic views learned by MLMs/GNNs. Fig. 3 depicts the sem2vec high-level pipeline. sem2vec first disassembles the input executable. Fig. 3(a) shows that for each assembly function $\mathcal{F}$, sem2vec performs under-constrained SE (USE) [69] from the entry point of $\mathcal{F}$ and gradually dissects the CFG of $\mathcal{F}$ into *tracelets*, representing continuous and short execution traces that are reachable from the function entry point via symbolic execution.[1] When traversing each tracelet, USE computes symbolic constraints and collects metadata regarding function calls. Fig. 3(b) shows a semantics-level feature sample for a tracelet. These features comprise symbolic constraints, external calls encountered on the tracelet, and call stack information. Details of tracelet-based USE are in Sec. 4.1.

To embed a tracelet's symbolic state, the key challenge is to comprehend symbolic constraints. Sec. 4.2 introduces methods to train a well-performing model, RoBERTa [57], which converts one symbolic constraint into an embedding vector. We select embeddings of K symbolic constraints and use HBMP [79] to compress them into a joint embedding of length L. K and L are user-configurable hyper-parameters (see hyper-parameter study in Sec. 6.6). We use one bit to encode the call stack and one-hot encoding to encode the external calls, both of which are concatenated with embeddings of symbolic constraints to build the tracelet embedding.

Tracelets of $\mathcal{F}$ form a connected graph $\mathcal{G}$ ("Tracelet Graph" in Fig. 3) where each node is a tracelet embedding. As shown in Fig. 3(c), we use GNNs to aggregate tracelet embeddings on G. This way, we generate the embedding of $\mathcal{F}$ (see details in Sec. 4.3).

Application Scope. To support security and code comprehension on legacy code, `sem2vec` is primarily designed to compute embeddings for 64-bit x86 executables. The prerequisite is disassembly; `sem2vec` analyzes disassembled machine code. `sem2vec` employs a commonly-used reverse engineering and SE engine, `angr` [75]. `angr` manifests very-high engineering quality in our usage, and we assume that disassembling is reliable. We also assume assembly functions are correctly recognized during reverse engineering [18]. Although our current focus is primarily on x86 64-bit executable (given its popularity), `angr` lifts machine code into platform-neutral representation for symbolic execution. Thus, `sem2vec` is able to analyze executables on other platforms (e.g., ARM) as long as `angr` can lift them. In evaluation, we measure the performance of `sem2vec` using cross-architecture settings by comparing executables on x86 with executables on aarch64 architectures; `sem2vec` achieves consistently encouraging accuracy. Also, `sem2vec` does not require symbols or debug information in executables, and thus stripped executables can also be processed.

4.1 Tracelet-Based USE

Alg. 1 presents our tracelet-based USE, where ***Traverse_CFG*** is the entry point. `sem2vec` performs USE from the entry basic block B_{entry} of the target function F (line 23). Subsequently, each time given a starting block B_0 (line 27), function ***Traverse_Tracelet*** traverses the derived tracelets maintained in $\mathcal{Q}$. When the current traversal has forked over MAX_STATE symbolic states (line 5), meaning $\mathcal{Q}$ is holding MAX_STATE tracelets starting from B_0, we finish the traversal and collect the symbolic states (see samples in Fig. 3(b)) of each tracelet.

4.1.1 ***Symbolic_Execution***.

We perform USE [69]: each time starting from a fresh basic block, ***Traverse_Tracelet*** initializes an empty symbolic state S_0 (line 3). When creating S_0, we represent the value of each register using free symbols (e.g., edx_0 in Fig. 3(b)), indicating that they have no associated constraint.

Symbolic_Execution performs SE on block B (lines 9–10), where we interpret each machine instruction and update the symbolic state S into S' accordingly. We create new free symbols to represent the values stored in memory cells in the case that such memory cells are loaded for the first time and the values stored in them are unknown. This subsumes data loading from function parameters, stack, heap, and global data. In addition to symbolic values of registers and memory locations, each symbolic state maintains the path constraint C (e.g., `eax < 0` in Fig. 3(b)) of the tracelet, denoting conditions that must be satisfied for execution to the current block B from the entry point B_0 of the current tracelet.

When function callsites are encountered on the path, we recursively inline callee functions if they are user-defined functions. Therefore, the path is expanded to subsume other user-defined functions. We also maintain

[1]Note that our definition of "tracelets" are different with TRACY [28]; please refer to Sec. 9 for discussion and comparison with prior works.

Algorithm 1 Tracelet-based symbolic execution.

```
1: function TRAVERSE_TRACELET($B_0$, $MAX\_STATE$)
2:     $B_0.has\_visited \leftarrow true$
3:     $S_0 \leftarrow$ INIT_SYMBOLIC_STATE($B_0$)
4:     $\mathcal{Q} \leftarrow \{S_0\}$
5:     while LEN($\mathcal{Q}$) < $MAX\_STATE$ do
6:         $\mathcal{Q}_{\text{next}} \leftarrow \varnothing$
7:         for each $S$ in $\mathcal{Q}$ do
8:             if $S.current\_block \neq$ NULL then                ▷ $S$ is not terminated.
9:                 $B \leftarrow S.current\_block$
10:                $S' \leftarrow$ Symbolic_Execution($S$)
11:                if HAS_CONDITIONAL_TRANSFER($B$) = $true$ then
12:                    $(S'_i, S'_j) \leftarrow$ FORK($S'$)              ▷ $S'$ has two successors.
13:                    $\mathcal{Q}_{\text{next}} \leftarrow \mathcal{Q}_{\text{next}} \cup \{S'_i, S'_j\}$
14:                else
15:                    $S'_{next} \leftarrow S'.next\_state$              ▷ $S'$ has one successor.
16:                    $\mathcal{Q}_{\text{next}} \leftarrow \mathcal{Q}_{\text{next}} \cup \{S'_{\text{next}}\}$
17:            else
18:                $\mathcal{Q}_{\text{next}} \leftarrow \mathcal{Q}_{\text{next}} \cup \{S\}$      ▷ Keep the terminated state.
19:        $\mathcal{Q} \leftarrow \mathcal{Q}_{\text{next}}$
20:    return CONVERT_STATES_TO_TRACELETS($\mathcal{Q}$)
21: function TRAVERSE_CFG($\mathcal{F}$, $MAX\_STATE$)
22:    $\mathcal{R} \leftarrow \varnothing$
23:    $\mathcal{B}_{\text{entry}} \leftarrow$ ENTRY_POINT($\mathcal{F}$)
24:    $\mathcal{B}_{\text{stack}} \leftarrow \varnothing$
25:    PUSH($\mathcal{B}_{\text{entry}}, \mathcal{B}_{\text{stack}}$)
26:    while EMPTY($\mathcal{B}_{\text{stack}}$) = $false$ do
27:        $B_0 \leftarrow$ POP($\mathcal{B}_{\text{stack}}$)
28:        if $B_0.has\_visited = false$ then
29:            $\mathcal{T} \leftarrow$ Traverse_Tracelet($B_0$, $MAX\_STATE$)
30:            $\mathcal{R} \leftarrow \mathcal{R} \cup \mathcal{T}$
31:            for each $T$ in $\mathcal{T}$ do                     ▷ Iterate all collected tracelets.
32:                if $T.next\_block \neq$ NULL then
33:                    PUSH($T.next\_block, \mathcal{B}_{\text{stack}}$)
34:    return $\mathcal{R}$
```

a call stack so that when encountering a `ret` instruction, we pop the latest caller function f and resume SE from the instruction next to the executed callsite in f. The chosen SE engine, `angr`, implements some C library functions. However, when the called C library function is not modeled by `angr`, we skip performing SE and use a free symbol to denote its return.

4.1.2 ***Traverse_Tracelet***. Starting from block B_0, ***Traverse_Tracelet*** performs a breadth-first search (BFS) on the CFG.

State Fork. Lines 11–13 in ***Traverse_Tracelet*** denote an important scenario, where `sem2vec` encounters a conditional control transfer at the end of block B on a tracelet. For conditional transfers, `sem2vec` forks the current symbolic state S' of a tracelet into two S'_i and S'_j w.r.t two successor blocks B_i and B_j, respectively. A condtional transfer increments the number of symbolic states by one. Suppose the control transfer predicate at the end of B is C_c, and the path constraint is C. Resuming SE on B_i yields path constraint $C_c \wedge C$ whereas resuming SE on B_j yields constraint $\neg C_c \wedge C$.

Maximal Symbolic States. Path explosion is a common issue when performing SE over CFG [16, 74]. Nevertheless, sem2vec stops symbolic execution whenever the number of symbolic states exceeds MAX_STATE (line 5). Therefore, our BFS traversal allows maximal MAX_STATE tracelets each time, thus mitigating path explosion. MAX_STATE is a hyper-parameter that is user-configurable. Before returning, we collect the final symbolic states and convert them to corresponding tracelets for use (line 20). An alternative to avoid the path explosion is *limiting the length of tracelets*, which is used by TRACY [28]. However, we found this method cannot tackle challenging comparison settings (e.g., binaries obfuscated by `control-flow flattening`) since obfuscation techniques frequently modify the structure and insert garbage code. The max length evaluated by TRACY is 5. A short tracelet likely contains little meaningful information of obfuscated code.

Tracelet Metadata. As shown in Fig. 3, when traversing a tracelet, we also collect metadata, including whether each tracelet ends within target function $\mathcal{F}$. Inspired by previous research [24], we also consider *external function callsites* as critical features. Holistically, compiler optimizations and obfuscations should not change external function calls to dynamically linked libraries [52]. Hence, we collect external callsites on each tracelet. These metadata contribute to the tracelet-level embedding, as shown in Sec. 4.2.

Terminating a Traversal. Tracelet traversal terminates when: 1) sem2vec has reached a `ret` instruction in function $\mathcal{F}$; 2) the underlying SE engine, angr, is unable to resolve a code pointer (e.g., an x86 indirect jump); or 3) the latest symbolic state fork increases the number of symbolic states into MAX_STATE. For the first two cases, we cannot locate successor blocks (lines 8 and 32 will be "false"). For the third case, we first check whether the call stack is empty. An empty call stack means that the successor block of a tracelet is still in function $\mathcal{F}$. If so, we push the successor block into $\mathcal{B}_{\text{stack}}$ (line 33). If not, we backwardly search the maintained call stack until we find the latest callsite in $\mathcal{F}$. We push the successor block of this callsite in $\mathcal{B}_{\text{stack}}$. Then, we re-run ***Traverse_Tracelet*** by popping one block B_0 from $\mathcal{B}_{\text{stack}}$ (line 27).

BFS or DFS. Readers may wonder why sem2vec implements the algorithm with BFS rather than depth-first search (DFS). We tentatively explored using DFS in our preliminary study, which was not desirable. The reason is that we seek to extract information from as many paths as possible to present a comprehensive view of the target function's semantics. If the analyzed function has reasonable complexity, we argue that both BFS and DFS can allow the SE engine to cover all of its paths smoothly. However, when the target function is substantially complex, a SE engine equipped with DFS may be frequently trapped in a lengthy path. Most of the efforts would be spent on such a single (or a few) deep path. As a result, when we have collected sufficient tracelets (or reaching timeout and terminate the analysis of the target function), most of the extracted information reflects only a few paths. Our observation shows that this would frequently undermine the accuracy of follow-up function matching, since other irrelevant paths may be analyzed for the function in comparison. Then, the covered paths in two functions are simply not matched, let alone their extracted symbolic constraints. In contrast, a SE engine equipped with BFS should manifest much better comprehensiveness, given that it prioritizes to cover as many paths as possible during the traversal. Notice that our SE engine will not re-visit a block (line 28), which also augments the path coverage from the implementation perspective.

Unreachable Path Pruning. As aforementioned, when sem2vec encounters a conditional path, it forks the symbolic state into two, implying both paths are reachable. However, because we keep track of path constraints C for each tracelet, sem2vec can use the constraint solver (angr uses Z3 [64]) to conduct on-the-fly unreachable path pruning. Specifically, given path constraint C and the current predicate C_c, we check the following two constraints:

$$\begin{aligned} &\text{(a)} \quad C \land \neg C_c \\ &\text{(b)} \quad C \land C_c \end{aligned} \tag{1}$$

Eq. 1(a) determines whether we can find solutions to execute the false branch ($\neg C_c$). If Z3 yields **unsat** ("no satisfiable solution"), it implies that the false branch is unreachable. Eq. 1(b) decides whether the true branch (C_c) is unreachable. sem2vec skips unreachable paths and refrains from creating a symbolic state.

Pruning unreachable code reduces the complexity of our analysis. More importantly, it helps to defeat obfuscation. In general, obfuscation methods tend to generate junk code to some extent [52]. Therefore, when pruning unreachable code, sem2vec becomes more resilient to obfuscation, as will be seen in Sec. 6.1.

We also clarify that detecting unreachable code by checking constraints in Eq. 1 is not *complete* but *sound*. The reason is that we launch USE to construct path constraints C on a tracelet until reaching B, which indeed omits all computation from main to this tracelet. Therefore, we may find satisfiable solutions which are indeed invalid when considering the entire path prefix and possible constraints over inputs/globals/memory cells from main to B. Therefore, if we are still unable to find a solution for any constraint in Eq. 1, it implies that the checked path *must* be unreachable.

4.1.3 ***Traverse_CFG****.* When finishing the traversal of tracelets $\mathcal{T}$ starting from block B_0, we collect the symbolic state of each tracelet for use. We also decide whether each $T \in \mathcal{T}$ has a successor block in $\mathcal{F}$ where we can re-run ***Traverse_Tracelet*** with a fresh symbolic state. All successor blocks are maintained in $\mathcal{R}$ as future inputs of ***Traverse_Tracelet***. ***Traverse_CFG*** proceeds until no new starting block can be obtained, meaning that all reachable blocks on the CFG of $\mathcal{F}$ have been covered. Also, we clarify that loops, though pervasively exist, should not be an obstacle in our analysis. Overall, we observe that the number of states when analyzing a loop can reach MAX_STATE easily, and since sem2vec with not re-visit a block, sem2vec will not be trapped in the loop and continue the analysis to the successor statements.

4.2 Tracelet Embedding

Fig. 3(b) illustrates the embedding of a tracelet: an embedding is formed by concatenating the embedding of symbolic constraints, the embedding of external callsites, and one bit denoting call stack information. For a collection of assembly functions, the number of invoked external functions (e.g., in standard glibc) is often limited, and we can build a vocabulary. We then apply one-hot embedding on the set of invoked external functions. In case the number of external functions is enormous, as a common tactic, users can compress this one-hot embedding design to an embedding of fixed length with a fully-connected (FC) layer. Call stack indicates whether the successor block of a tracelet belongs to the target function $\mathcal{F}$. Sec. 4.1.1 states that we inline all encountered callees during USE; therefore, some tracelets end with blocks that are not in $\mathcal{F}$. We use one bit to encode this information.

Performing representation learning over symbolic constraints is generally obscure. It necessitates understanding the semantic-level equivalent/inequivalent relations rather than syntactical features of constraints. To our knowledge, sem2vec is the first attempt to perform representation learning directly over symbolic constraints. Each input/output constraint or path constraint is converted into an embedding vector in a unified manner detailed as follows.

Preprocessing Symbolic Constraints. We first flatten a symbolic constraint through an in-order traversal to generate a sequence of tokens for each constraint. Constants in symbolic constraints are typically very sparse, ranging from 0 to 2^{64}. Hence, directly consuming raw constants during representation learning can result in enormous numbers of low-frequency words and frequently encounter the infamous out-of-vocabulary (OOV) problem, which may undermine the subsequent learning process. Nonetheless, our empirical findings show that by converting constants into logarithmically normalized forms, the embedding quality can progressively grow. Hence, a constant c is converted to $2^{\lfloor \log c \rfloor}$ (when $c > 0$) or 0 (when $c = 0$). Intuitively, logarithmic normalization can map constants into a small set of distinct values while keeping many important constants, e.g., $0, 1, 2^{64}$, distinguishable to the model. In contrast, standard "constant normalization" may lose considerable expressiveness because it often converts any constant values into a unified type symbol (e.g., `INT`, `STRING`) [19].

Pre-training Using Whole Word Masking (WWM). We employ WWM to train an embedding model in an unsupervised manner. Given a preprocessed symbolic constraint C, we *randomly* mask 15% of tokens x_m in s and obtain C', e.g., $(x_1, x_2, x_3, \cdots, x_n) \rightarrow (x_1, \texttt{[MASK]}, x_3, \cdots, x_n)$. We then leverage RoBERTa [57], a representative instance of the BERT family [29], to perform pre-training on these masked symbolic constraints. In particular, we let RoBERTa perform a multi-class prediction task to recover the masked tokens x_m based on the context in C'. The cross-entropy loss (L_1) on the prediction is employed as the training objective as follows:

$$L_1 = -\sum \log P(x_m \mid C') \tag{2}$$

where x_m is the original token that is masked in C. C' denotes the masked context, which is the masked symbolic constraint here. The trained WWM can convert C into an embedding vector v by using the output of mean pooling over token-level embeddings.

Pre-training with Siamese Network. The aforementioned process converts symbolic constraints into numeric vectors. However, symbolic constraints carry richer information than plaintext. In particular, *semantics* encoded in symbolic constraints have not yet been considered. Recent studies [70] from the NLP community also indicate that standard BERT-like models may primarily use tokens for embedding, instead of extracting a more holistic understanding from entire sentences (here "sentences" are symbolic constraints).

We construct a Siamese network [70] to train the RoBERTa model by matching symbolic constraints that come from the same line of source code. Therefore, we improve the embedding quality of syntactically-distinct symbolic constraints with identical semantics, which likely occur due to compiler optimizations or obfuscations.

We first prepare a collection of constraint pairs (f_1, f_2) by performing SE on assembly code corresponding to the same line of source code; these assembly codes are generated by compiling the same program using different compilation settings (i.e., `gcc -O0`, `gcc -O3`). Therefore, each constraint pair contains two syntactically-distinct constraints but with identical semantics. We also prepare a collection of inequivalent constraint pairs by randomly selecting and pairing constraints. We use programs from Linux `coreutils` to form the corpus.

Our embedding model takes these equivalent/inequivalent pairs of symbolic constraints as inputs. The model first takes a constraint pair (f_1, f_2) and yields the corresponding pair of embedding vectors (v_1, v_2) from RoBERTa. Then, the model leverages cosine similarity as the objective and trains RoBERTa to force it to group constraints that come from the same line of source code. RoBERTa is also traind to distinguish constraints with inequivalent semantics. The loss function is defined as follows:

$$L_2 = -\sum (l - \frac{v_1 \cdot v_2}{|v_1| \times |v_2|})^2 \tag{3}$$

where l is the label denoting whether the constraint pair is equivalent ($l = 0$) or not ($l = 1$) and $\frac{v_1 \cdot v_2}{|v_1| \times |v_2|}$ computes the cosine similarity between the two vectors. This way, our constraint embedding model becomes gradually more resilient toward different syntactical forms of constraints.

Selecting Representative Constraints. Each tracelet's symbolic state contains several constraints representing input/output relations and the path constraint over the tracelet. Here, we select K representative constraints from the symbolic state: these constraints include one path constraint, and input/output constraints over $K - 1$ registers. Our preliminary experiments show that the longest symbolic constraints are usually informative enough. Hence, `sem2vec` selects the embedding vectors corresponding to the longest $K - 1$ symbolic constraints. K is a user-configurable hyper-parameter.

Given K embedding vectors, we further use HBMP [79], a popular recurrent neural network (RNN) model that shows encouraging performance in learning distributed representations, to compress K vectors into one vector of length L. L is a hyper-parameter. As in Fig. 3, this vector is concatenated with embeddings of external calls and call stack to form the tracelet's embedding V_{tracelet}.

4.3 CFG-Level Embedding

As aforementioned, tracelets of function $\mathcal{F}$'s CFG forms a connected graph $\mathcal{G}$. Since we have computed an embedding vector for each tracelet (i.e., a node of $\mathcal{G}$), the next step is to compute an embedding of $\mathcal{G}$, denoting the embedding of $\mathcal{F}$. We reuse a well-performing graph embedding pipeline proposed in `BinaryAI` [93] to compute $\mathcal{G}$'s embedding. In particular, we use the GGNN [56] message passing scheme to iteratively update each embedding vector V_{tracelet} based on its neighboring nodes. Then, we use Set2Set [82], a graph pooling scheme, to aggregate each vector V_{tracelet} into a unified embedding vector $\mathcal{V}$. $\mathcal{V}$ is the embedding of $\mathcal{G}$. We clarify that in the current implementation, $|\mathcal{V}|$ is 256, the same as the `BinaryAI`'s function embedding size. `BinaryAI` ships with circle loss [78], a common loss function, to form the learning objective for training.

Design Decision. `BinaryAI` uses BERT [29] to produce token-level embedding, whose output is fed to HBMP for block-level embedding and then for graph embedding. Therefore, by replacing `BinaryAI`'s token-level embedding with `sem2vec`, we present an *ablation evaluation* to better understand the strength of learning from semantics. Sec. 6.1.1 empirically shows that compared with `BinaryAI`, learning from semantics is generally more accurate.

Moreover, it is feasible to adopt some other graph-level embedding methods [66, 89, 97], which might potentially enhance `sem2vec` to a certain extend. In Sec. 6.4, we replace the graph embedding modules of `BinaryAI` with another graph embedding model, `Gemini` [92]; we constantly achieve promising results and outperform the state-of-the-art (SOTA) model `PalmTree` [54], which is also based on `Gemini`. Overall, `sem2vec` aims to provide a practical embedding pipeline for x86 binary code. Designing novel neural embedding models is *not* our focus. Prior GNNs are sufficient to compute high-quality embeddings.

5 IMPLEMENTATION

`sem2vec` is primarily written in Python with about 11K LOC (see its codebase at [11]). As previously clarified, we adopt a popular binary code analysis platform, `angr` [75], for disassembling and SE. The current implementation disassembles 64-bit x86 executables in the ELF format. We implement RoBERTa with Transformers [88] and train it with the official implementation of Sentence-BERT [70].

`sem2vec` needs to be bridged with graph-level embedding models to process assembly functions. We have clarified the implementation details of deploying `sem2vec` on `BinaryAI` in Sec. 4.3. To illustrate the generalizability of `sem2vec`, we deploy `sem2vec` and the SOTA model, `PalmTree` [54], on `Gemini` for evaluation and comparison in Sec. 6.4. [54] has shown how to deploy `PalmTree` on `Gemini`: `PalmTree` generates a basic block embedding vector by averaging embeddings of all instructions in the block with mean pooling. Accordingly, we average the embeddings of four input-output constraints and one path constraint to get a tracelet embedding vector.

6 EVALUATION

Programs. We use datasets Linux `coreutils` (verison 8.32), `binutils` (verion 2.36), `diffutils` (verison 3.7), `findutils` (verison 4.8), `OpenSSL` (verison 1.1.1h), `libtomcrypt` (version 1.18.2), `libgmp` (version 6.2.1), `zlib` (version 1.2.12), and `rapidjson` (version 1.1.0) for the evaluation. Existing works [13, 15, 30, 55, 58, 92, 99] also primarily evaluated these datasets or a subset of them. `coreutils`, `binutils`, `diffutils` and `findutils` consist of common Linux utilities with diverse functionalities such as textual processing and system management. In total, these datasets constitute **116,941,424 pairs of assembly functions** (see breakdowns in Table 1) and **713,209,256 assembly instructions** in total. In addition, we rebuild a vulnerability dataset used in [28, 30], which contains vulnerable samples in real-world software such as FFmpeg and Bash. In Sec. 6.2, we show `sem2vec` can augment vulnerability search over this dataset.

Compiler. We use GNU `gcc` ver. 7.5.0 and `clang` ver. 4.0.1. We benchmark cross optimization evaluation using three optimization levels (-O0, -O2, -O3) of these two compilers.

Obfuscation Schemes. We evaluate a widely-used obfuscation framework, the LLVM Obfuscator [50] (ver. 4.0.1). LLVM Obfuscator (referred to as `ollvm` in this paper) provides the following three popular obfuscation schemes: 1) `-sub`, denoting instruction substitution that replaces simple operations (e.g., addition) with semantics-equivalent but syntax-level more complex formats, 2) `-bcf`, denoting bogus control flow that inserts opaque predicates to protect conditional branches. Such predicates are usually difficult to evaluate until runtime [52], and 3) `-fla`, denoting control-flow flattening that changes the structure of the original control flow into a "flattened" structure. The execution flow is chained by a C switch statement to iterate basic blocks [52].

`ollvm` is integrated into the LLVM framework. Hence, three obfuscation methods can be enabled together when compiling source code using LLVM. We thus prepare the 4th obfuscation scheme by enabling all three methods together when compiling a program. This scheme is referred to as `-hybrid`.

Metrics. We compute the standard top-k accuracy and the Normalized Discounted Cumulated Gain (NDCG) scores [46]. We compile a program into two versions of executables Bin_1 and Bin_2 using different compilation/obfuscation settings. For each function f_1 in Bin_1, we iteratively compare it with all functions in Bin_2. The correct match of f_1 should be an assembly function f_{target} in Bin_2 sharing an identical function name with f_1. Then, the top-k accuracy is computed by checking whether f_{target} appears in the top-k comparison pairs ranked by their similarity scores. The similarity score of two assembly functions is from the cosine distance of their embeddings. The NDCG scores are also computed based on the vectors of similarity scores. Since functions that have a single tracelet are usually trivial, `sem2vec` considers functions with more than one tracelets, For `ASM2VEC`, `PalmTree`, and `BinaryAI`, we consider functions with at least five blocks, which is taken by `ASM2VEC`.

Baseline. We compare `sem2vec` with four SOTA DNN-based binary embedding tools, `PalmTree` [54], `ASM2VEC` [30], `BinaryAI` [93, 94], and `SAFE` [61]. We also compare `sem2vec` with industrial-strength binary matching tool, `BinDiff` [1], which features a classic graph isomorphism-based binary code comparison. As mentioned in Sec. 4.3, `BinaryAI` conducts assembly function embedding by computing basic block embeddings with HBMP [79] and then conducting graph embedding with GGNN/Set2Set [56, 82].

`PalmTree` [54], denoting the SOTA work, provides a novel language model for x86 machine instruction embedding. `PalmTree` features a flexible self-supervised training procedure over unlabeled assembly code. Its instruction representation is shown as effective over popular downstream tasks like code similarity analysis, function prototype inference and static analysis. `PalmTree` focuses on computing a high-quality embedding of machine instructions. It uses mean pooling for basic block-level embedding, and `Gemini` [92] for control graph (function)-level embedding. We follow its paper to equip `PalmTree` with mean pooling for basic block embedding. As for graph-level embedding, we consider two configurations by using `BinaryAI` and `Gemini`. These two implementations are referred to as $\texttt{PalmTree}_B$ and $\texttt{PalmTree}_G$ in the evaluation, respectively. Note that `PalmTree` was an instruction embedding generation tool which is not fine-tuned; we do the same. That is, $\texttt{sem2vec}_B$ and $\texttt{sem2vec}_G$ employ pre-trained RoBERTa as symbolic constraint embedding tool **without** fine-tuning.

`ASM2VEC` generates assembly function embedding using an extended PV-DM MLM [53] and GNNs. We set up its official client, which requires IDA-Pro. However, `ASM2VEC` does not provide a pre-trained model to reproduce its reported results. We follow the description in its paper to build the indexing dataset with binaries (e.g., `coreutils`) compiled using -O0. We use all its default settings. `SAFE` is based on self-attentive neural networks. It treats the instruction sequence as a natural language corpus. `SAFE` applies a gated recurrent unit (GRU) RNN on the instruction sequence; it then uses the attention mechanism to process all GRU hidden states to focus on the portion of representative binary codes. We get the `SAFE` results in Table 1 with the officially-released model.

Processing Time of SE. Tracelet-based SE can be launched in parallel. sem2vec currently employs 30 threads. For binary code produced by different compilation settings, sem2vec spends 76.6 to 120.8 CPU seconds to launch SE on average per function; obfuscation notably complicates executable, and the processing time is accordingly increased with the extent of obfuscation.

The processing time of sem2vec's SE module, approximately one to two minutes per function, is not insignificant. Overall, we observe that SE accounts for the majority of the time cost of sem2vec. Given that said, we like to highlight that it is technically feasible to *parallelize* the SE of each assembly function. In fact, the real time (not CPU time) required to execute a single binary in our test datasets is typically *less than one hour*, indicating a reasonable cost. In addition, we clarify that launching SE for each function is a one-time endeavor. Once the embedding of a function has been generated, it can be saved to disk, and subsequent comparisons against this function are performed quickly.

From the implementation perspective, sem2vec is implemented on top of angr, one state-of-the-art SE engine with active community support and documents. We view this design choice is reasonable, and aligned with many existing works in this field. Nevertheless, sem2vec is not specific to angr, and given that angr is implemented with Python, it is deemed as slower than C/C++. At this step, we tentatively benchmarked an advanced symbolic execution engine, QSYM [95], which is written in C/C++ and particularly optimized for analysis speed. In short, we find that QSYM is about 24 times faster than angr, which is consistent with [68]. We conclude, therefore, that an effective, advanced SE engine can presumably enhance our SE process greatly. However, we implement the prototype of sem2vec with angr because QSYM was designed primarily for concolic execution and it is difficult to expand it for under-constrained SE. That is, it can only analyze programs from the executable entry point and is unable to start from arbitrary program points and launch under-constrained SE by modeling all program states using symbols. We leave this as future work to replace angr with advanced, speedy symbolic engines that support symbolic execution. Also, our evaluation in this paper has shown that sem2vec can scale to large datasets (e.g., OpenSSL), and therefore, we conclude that sem2vec manifests reasonable scalability toward real-world, common binary samples.

Training Dataset. To form the training dataset of sem2vec, we compile coreutils and binutils programs using gcc and clang with two optimization levels (-O0, -O3), in total, four compilation settings. This setup generates in total $4159 \times 2 \times 2$ functions. As a common step, we randomly split this collection of functions into nine-fold training and one-fold test datasets. That is, the 1 and 6 comparison settings in Table 1 are using the one-fold test datasets. Data in all the other comparison settings are *not* used for training. We train $\texttt{PalmTree}_B$ and BinaryAI with the same setup.

Training Time. Training is done on a server with two Intel Xeon Platinum 8255C CPUs and one Tesla V100-SXM2 32GB. We report that the constraint embedding phase is trained within 40 hours. It takes 4 hours to train BinaryAI and 6 hours to train Gemini.

Table 1. Top-1/top-3/top-5 accuracy and NDCG score evaluation in terms of different settings. We also comparison with the SOTA (DNN-based) binary code matching search tools. Note that `sem2vec`$_B$ denotes `sem2vec` + `BinaryAI` and `PalmTree`$_B$ denotes `PalmTree` + `BinaryAI`. To enhance readability, for each comparison setting, we mark the 🟩 highest top-1/NDCG scores and the 🟦 second-highest top-1/NDCG scores . ⬛ Specious results denote results that, to our best knowledge, need to be excluded to deliver a *fair* comparison.

ID	coreutils (444889 × 6 pairs of Function × Function)											
	S^1	`sem2vec`$_B$		`PalmTree`$_B$		`BinaryAI`		`ASM2VEC`		`SAFE`		`BinDiff`
1	S_1	78.5 /90.9/93.0	.892	63.6/78.9/82.3	.783	74.0 /77.6/80.3	.816	63.3/77.1/80.7	.763	22.2/31.7/37.7	.422	10.9/NA/NA	NA
2	S_2	78.7 /89.8/92.5	.886	55.3/71.5/77.2	.724	73.9 /75.5/79.4	.800	53.7/68.7/75.1	.694	23.8/36.8/43.1	.431	9.69/NA/NA	NA
3	S_3	74.5 /87.4/90.6	.859	50.1/67.8/74.4	.690	69.3 /74.4/79.1	.781	19.3/32.7/37.4	.342	22.9/35.3/40.7	.421	11.0/NA/NA	NA
4	S_4	73.9 /87.7/90.0	.860	19.6/33.5/39.0	.424	33.7 /46.8/52.4	.529	17.3/26.6/31.8	.300	13.8/20.4/25.0	.337	10.5/NA/NA	NA
5	S_5	60.6 /75.6/79.5	.753	1.38/3.18/4.98	.175	35.2 /49.9/55.8	.555	11.5/20.0/25.7	.242	11.0/15.1/18.9	.291	8.17/NA/NA	NA
6	S_6	52.2 /71.2/75.9	.700	0.83/2.07/4.15	.163	13.3/21.2/26.8	.329	6.95/11.4/13.7	.141	5.96/8.27/10.1	.236	13.6 /NA/NA	NA
ID	binutils (4251844 × 6 pairs of Function × Function)											
7	S_1	74.8 /88.8/92.6	.865	62.4/76.6/80.4	.764	74.7 /87.3/89.6	.867	64.0/78.5/83.0	.778	19.3/26.9/32.8	.383	9.30/NA/NA	NA
8	S_2	74.8 /89.2/92.6	.869	62.4/77.3/82.0	.770	77.6 /89.6/92.0	.896	59.7/75.2/80.3	.748	18.3/27.8/34.6	.398	10.2/NA/NA	NA
9	S_3	68.3 /84.5/88.9	.823	52.3/68.2/74.2	.696	72.3 /87.1/89.9	.870	56.2/74.5/79.0	.724	17.3/25.5/32.6	.384	9.50/NA/NA	NA
10	S_4	67.2 /83.4/88.3	.814	18.8/28.4/33.5	.386	40.0 /52.9/58.0	.574	50.3/67.4/72.4	.663	9.95/14.6/18.9	.296	7.23/NA/NA	NA
11	S_5	44.2 /59.4/65.1	.623	0.49/1.61/2.79	.143	36.8 /54.3/60.6	.576	36.5/51.8/58.5	.535	8.59/10.5/13.7	.249	10.6/NA/NA	NA
12	S_6	33.5 /46.3/52.3	.520	0.39/1.13/2.01	.134	11.8/18.9/22.9	.302	17.5 /31.2/37.6	.342	5.98/7.30/9.17	.209	10.1/NA/NA	NA
ID	OpenSSL (14160169 × 6 pairs of Function × Function)											
13	S_1	61.9 /77.1/81.7	.770	34.1/49.5/56.8	.548	62.5 /76.5/81.2	.796	46.8/56.9/60.6	.579	27.2/36.5/44.3	.463	82.4/NA/NA	NA
14	S_2	54.1 /71.1/77.3	.738	28.7/42.4/49.0	.495	54.4 /70.3/75.9	.743	46.2/55.9/59.4	.567	24.7/34.0/41.6	.451	82.3/NA/NA	NA
15	S_3	50.1 /65.9/71.9	.693	27.8/41.5/47.9	.488	52.1 /61.9/66.6	.713	42.3/51.8/55.8	.532	24.6/32.7/39.5	.445	82.0/NA/NA	NA
16	S_4	49.7 /66.1/71.7	.693	10.9/17.1/20.3	.282	26.9/36.8/41.0	.393	28.9 /36.3/40.0	.384	11.6/14.5/17.1	.299	79.0/NA/NA	NA
17	S_5	31.0 /44.5/49.8	.506	2.95/3.49/3.92	.148	24.0/37.3/43.2	.448	25.2 /33.0/36.7	.352	11.9/14.5/17.1	.251	81.4/NA/NA	NA
18	S_6	23.8 /35.1/40.4	.434	2.58/3.32/3.62	.140	8.60 /13.6/16.3	.243	5.1/6.88/8.38	.088	6.1/7.99/9.07	.211	79.8/NA/NA	NA
ID	diffutils (30276 × 6 pairs of Function × Function)											
19	S_1	74.5 /86.7/90.7	.833	64.5/80.1/86.7	.799	85.7 /91.3/93.4	.915	58.6/74.9/82.7	.753	39.6/54.1/61.2	.583	24.4/NA/NA	NA
20	S_2	77.6 /89.7/92.0	.847	58.4/74.2/78.7	.745	83.7 /90.9/93.8	.912	49.4/69.2/74.4	.674	35.6/54.9/63.4	.591	18.7/NA/NA	NA
21	S_3	74.7 /85.6/92.0	.812	53.9/66.9/70.8	.700	78.5 /90.9/92.3	.897	43.6/69.8/76.2	.643	35.3/52.2/62.4	.585	21.0/NA/NA	NA
22	S_4	76.4 /87.4/92.0	.841	29.2/39.9/46.6	.505	53.6 /63.6/68.4	.686	41.0/57.1/64.0	.598	20.3/33.4/40.3	.467	21.5/NA/NA	NA
23	S_5	55.7 /73.0/80.5	.741	3.37/6.74/10.7	.244	51.7 /63.2/70.8	.741	29.1/51.4/57.5	.506	18.1/22.7/27.9	.396	19.8/NA/NA	NA
24	S_6	53.4 /70.1/75.3	.668	1.69/5.06/6.74	.215	25.4 /39.7/45.9	.463	15.5/25.6/31.9	.309	11.5/14.2/17.6	.329	20.6/NA/NA	NA
ID	findutils (96100 × 6 pairs of Function × Function)											
25	S_1	72.0 /85.8/89.7	.845	59.6/76.4/82.0	.760	73.9 /81.8/84.0	.821	45.9/60.2/66.0	.629	32.9/45.1/52.5	.512	15.1/NA/NA	NA
26	S_2	76.5 /89.0/93.5	.863	51.0/66.6/74.1	.693	70.8 /81.4/83.7	.841	48.5/63.8/70.0	.660	31.7/47.9/53.4	.530	13.8/NA/NA	NA
27	S_3	69.4 /86.8/91.9	.834	48.6/64.5/73.1	.678	67.0 /82.5/85.4	.825	44.1/60.5/68.3	.618	29.7/44.0/51.1	.516	13.6/NA/NA	NA
28	S_4	75.8 /88.1/92.9	.859	21.7/36.2/43.4	.451	39.5 /40.7/51.3	.603	34.5/56.0/62.9	.556	18.6/29.5/34.4	.404	18.0/NA/NA	NA
29	S_5	61.9 /74.5/81.3	.770	2.07/7.93/11.4	.225	37.2 /53.3/61.6	.664	20.6/36.0/45.1	.407	13.9/19.1/23.1	.335	17.5/NA/NA	NA
30	S_6	59.0 /75.2/81.0	.732	1.38/5.17/7.59	.202	16.3/24.4/30.4	.407	12.6/22.2/28.3	.275	9.40/13.3/16.1	.291	21.4 /NA/NA	NA

ID													
ID	libtomcrypt (283024 × 6 pairs of Function × Function)												
31	S_1	59.9/77.0/82.2	.768	52.9/77.2/85.9	.745	53.7/57.4/66.7	.650	19.8/35.0/41.0	.391	11.2/23.1/32.1	.362	97.2/NA/NA	NA
32	S_2	59.0/76.7/82.0	.765	54.9/79.3/86.6	.762	54.0/61.7/68.1	.666	23.7/41.6/52.4	.486	11.5/25.7/34.0	.371	99.0/NA/NA	NA
33	S_3	49.3/67.0/75.5	.688	47.8/71.7/80.4	.706	49.2/58.1/64.7	.635	20.0/36.2/46.0	.439	12.7/26.2/32.4	.372	98.7/NA/NA	NA
34	S_4	56.6/72.3/78.2	.740	23.4/40.6/51.0	.486	18.0/27.8/33.7	.383	15.9/30.3/40.4	.371	7.87/16.5/21.6	.313	98.1/NA/NA	NA
35	S_5	29.8/45.1/48.1	.504	2.14/6.06/10.2	.211	23.6/37.7/45.0	.468	15.8/28.8/37.0	.369	5.60/12.1/16.9	.269	98.3/NA/NA	NA
36	S_6	26.0/39.5/47.5	.475	1.43/3.57/6.42	.186	5.50/11.2/16.2	.254	7.69/15.2/20.4	.193	3.18/6.96/10.7	.216	98.4/NA/NA	NA
ID	zlib (8836 × 6 pairs of Function × Function)												
	S[1]	sem2vec$_B$		PalmTree$_B$		BinaryAI		ASM2VEC		SAFE		BinDiff	
37	S_1	77.4/88.7/91.9	.867	55.8/74.0/83.1	.751	74.9/87.1/90.0	.855	27.9/42.3/52.9	.494	27.7/41.5/52.1	.519	99.1/NA/NA	NA
38	S_2	71.0/87.0/90.0	.835	70.1/89.6/93.5	.847	69.2/82.6/87.1	.816	21.3/41.5/51.1	.470	25.5/47.9/61.7	.554	97.3/NA/NA	NA
39	S_3	62.3/81.2/85.5	.782	63.6/79.2/87.0	.801	64.0/79.6/84.6	.787	17.0/35.1/43.6	.407	29.8/51.1/57.4	.561	97.0/NA/NA	NA
40	S_4	63.8/85.5/92.8	.820	35.1/62.3/71.4	.621	33.0/47.7/53.8	.544	16.4/25.5/27.2	.291	23.4/38.3/50.0	.498	88.5/NA/NA	NA
41	S_5	42.0/68.1/72.5	.653	7.79/19.5/27.3	.340	32.6/48.4/55.9	.553	13.3/20.0/29.5	.286	12.8/27.7/36.2	.391	93.6/NA/NA	NA
42	S_6	34.8/58.0/62.3	.586	5.19/16.9/19.5	.314	15.2/25.1/30.1	.369	7.89/11.4/16.7	.180	9.57/18.1/24.5	.336	91.7/NA/NA	NA
ID	libgmp (210681 × 6 pairs of Function × Function)												
43	S_1	54.0/74.1/79.1	.727	33.6/50.8/61.3	.570	74.9/87.1/90.0	.855	20.8/26.2/30.0	.306	27.1/39.1/46.2	.479	94.8/NA/NA	NA
44	S_2	52.9/65.6/70.8	.688	33.1/50.5/56.5	.556	69.2/82.6/87.1	.816	18.2/22.4/24.9	.264	30.6/39.1/45.1	.494	95.3/NA/NA	NA
45	S_3	47.3/62.7/68.8	.657	32.3/46.2/54.3	.541	64.0/79.6/84.6	.787	16.0/20.7/21.9	.231	29.4/37.7/45.8	.486	94.8/NA/NA	NA
46	S_4	53.6/65.8/70.4	.696	15.1/25.5/33.9	.379	33.0/47.7/53.8	.544	14.1/16.3/18.8	.190	24.5/29.4/33.5	.416	93.5/NA/NA	NA
47	S_5	41.4/52.3/58.2	.587	1.34/3.76/4.57	.181	32.6/48.4/55.9	.553	14.5/17.8/19.0	.198	20.8/23.9/27.1	.360	94.6/NA/NA	NA
48	S_6	33.8/46.2/50.1	.521	1.08/3.23/4.30	.175	15.2/25.1/30.1	.369	11.0/13.0/14.5	.142	18.0/20.1/23.0	.320	94.0/NA/NA	NA
ID	rapidjson (13225 × 2 pairs of Function × Function). rapidjson is a C++ library.												
49	S_7	33.0/51.3/62.6	.581	17.2/31.1/38.5	.414	20.2/34.1/40.3	.442	7.08/13.3/14.2	.186	18.2/24.8/36.5	.408	6.38/NA/NA	NA
50	S_8	37.4/53.9/67.8	.601	19.7/30.3/41.0	.444	20.9/32.6/41.1	.437	0.97/9.71/10.7	.101	12.6/26.8/34.6	.393	12.8/NA/NA	NA

[1] Settings. S_1 denotes the comparison between gcc -O0 and gcc -O3. S_2 denotes the comparison between gcc -O0 and clang -O3. S_3, S_4, S_5, S_6 denotes the comparisons between gcc -O0 and ollvm -O3, with -sub, -bcf, -fla and -hybrid obfuscations respectively. S_7 denotes the comparisons between g++ -O0 and g++ -O3. S_8 denotes the comparisons between g++ -O0 and clang++ -O3.

6.1 Evaluation Results

Table 1 reports the evaluation results using different datasets in cross-compiler, cross-optimization, and obfuscation settings. As aforementioned, we use four obfuscation schemes provided by ollvm. Given that binary code with more intensive optimizations appears less similar to the un-optimized versions, we conduct a cross-optimization evaluation with the most challenging setup, i.e., comparing un-optimized (-O0) code with fully optimized (-O3) code.

Overall, sem2vec manifests high accuracy across nearly all comparison settings. Out of in total 50 challenging comparison settings, sem2vec yields the highest top-1 scores for 37 comparisons. For the rest 13 (50-37) comparison settings, sem2vec has the second-highest top-1 scores of 12 settings. We thus interpret the overall results as highly encouraging, illustrating the high accuracy and robustness of sem2vec under different challenging settings over different real-world datasets.

For the OpenSSL evaluation, we exclude the results of BinDiff to enable a *fair comparison* with other tools. The reason is that BinDiff will simply match two functions when they have identical function names. OpenSSL has a shared library whose export table, after stripping, still contains names of most functions. Therefore, BinDiff takes advantage of these function names and achieves a high accuracy. In Table 1, evaluation using other datasets with comparable complexity shows that BinDiff is obviously less accurate than modern DNN-based approaches.

we find that cross-compiler/optimization settings (ID 2, 8, 14, 20, 26, 32, 38, 44, and 50 in Table 1), manifest generally promising accuracy. Note that gcc and clang may frequently use distinct strategies in optimization and code generation phases [71]. Code optimizations, as illustrated in our motivating example (Fig. 2), also largely complicate the CFG. In particular, while top-1 accuracy is generally below 80%, sem2vec has a higher probability (i.e., over 90% for comparisons in ID 2, 8, 20, 26 and 38 in Table 1) of placing the correct matches in the top-5 ranked candidates. In contrast, Table 1 shows that when facing such cross-compiler and cross-optimization settings, prior tools may perform worse. We compare sem2vec with prior works in Sec. 6.1.1.

Heavy obfuscation schemes like -fla can largely complicate the control flow graphs, which imposes great difficulty for function matching. Compared with the non-obfuscation cases (ID 2, 8, 14, 20, 26, 32, 38, 44, and 50 in Table 1), sem2vec exhibits lower top-1 accuracy for comparisons involving -fla. For instance, top-1 accuracy using binutils is reduced to 44.2%. Nevertheless, the top-5 accuracy is largely improved to 65.1%. As expected, -hybrid denotes the most difficult setting. We found that this strategy considerably mutates the program CFG. We discover that multiple obfuscation schemes, when being used together, can yield synergistic effects, given that one scheme, e.g., -bcf, complicates the CFG and introduces more basic blocks. Therefore, consequence schemes like -fla can flatten a CFG with more blocks. In some circumstances, -hybrid obscures the analysis conducted by sem2vec, and we give further studies of erroneous matchings in Sec. 8. However, sem2vec outperforms other programs on all obfuscation strategies, including -hybrid, as other tools report significantly lower accuracy. Sec. 6.1.1 discusses these findings.

rapidjson is a C++ library and we failed to compile it when the obfuscations are enabled. Hence, we only provide the comparison results of cross-optimization (ID 49) and cross-compiler/optimization (ID 50). The accuracies of all works on these two settings are significantly lower than that of other normally compiled settings (ID 1, 2, 7, 8, 13, 14, 19, 20, 25, 26, 31, 32, 37, 38, 43, 44). We interpret the decrease is primarily caused by the difference of C++ and C code. Note that our model and all the baseline models are trained using binaries compiled from C source code. In contrast, binaries compiled from C++ source code often have significantly different external calls and structures, which impede the comparison of all models. Nevertheless, the performance of sem2vec$_B$ is still much better than other works. This illustrates the advantage of using symbolic execution to primarily extract semantics constraints for learning and comparison.

6.1.1 Comparison with Prior Works. Table 1 compares sem2vec with other works: BinDiff only provides top-1 matching for comparison; its performance largely falls behind other modern DNN-based tools except for highly obfuscated cases (e.g., setting 5), where nearly all DNN-based methods (except sem2vec) yield low accuracy. SAFE also shows notably less top-*k* accuracy compared with other DNN-based approaches.

We clarify that some DNN-based tools are also stated to extract a certain level of program "semantics" (using their own terminology) for comparison. For example, ASM2VEC and BinaryAI extract program lexical relations between tokens (referred to as "semantics" in their papers) and use NLP models to extract a relatively robust representation. Overall, it is shown that these works facilitate a reasonable enhancement compared with BinDiff, which features classic graph-level similarity comparison. Nevertheless, Table 1 illustrates that their extracted "semantics" are still *shallow*, manifesting lower obfuscation resilience than sem2vec.

Ablation Evaluation between sem2vec$_B$ and BinaryAI. As introduced in Sec. 4.3, sem2vec$_B$ and BinaryAI share the *same* graph-level embedding implementation. Therefore, comparing sem2vec$_B$ and BinaryAI forms an ablation evaluation that shows the key strength of sem2vec$_B$, namely, its ability to learn much more robust embeddings. In fact, by replacing the token-level embedding module in BinaryAI with tracelet-level embeddings computed by sem2vec, we have observed notable accuracy improvement: sem2vec$_B$ outperforms BinaryAI for 20 comparison settings in Table 1, particularly on challenging comparison settings involving -bcf and -fla. For the rest settings, sem2vec$_B$ also manifests close performance with BinaryAI. As aforementioned, BinaryAI is trained in an *end-to-end* manner, whereas `sem2vec`$_B$ uses the pre-trained constraint embedding generator without fine-tuning. This indicates the potentials of fine-tuning `sem2vec`$_B$ to improve the accuracy further.

Ablation Evaluation between `sem2vec`$_B$ and `PalmTree`$_B$. `PalmTree` leverages the well-established BERT [29] language model but is pre-trained with new learning objectives focusing on the inherent characteristics of assembly language. It is shown that the internal formats, control flow dependency, and data flow dependency of assembly instructions can be effectively captured by `PalmTree` when computing embeddings [54]. This illustrates the strength of tailored assembly instruction-level embedding techniques. Nevertheless, it is shown that `sem2vec`$_B$ constantly suppresses `PalmTree`$_B$. We attribute the encouraging results to the precise symbolic constraints captured by `sem2vec`. We note that in addition to Table 1 comparing `sem2vec`$_B$ and `PalmTree`$_B$, `sem2vec`$_G$ also outperforms `PalmTree`$_G$, as will be presented in Sec. 6.4.

6.1.2 Obfuscation Resilience Analysis. The above comparison depicts the key strength of `sem2vec`. We now analyze its resilience in detail and compare it with other works.

Instruction Replacement. `ollvm` enables instruction-level obfuscation via the `-sub` option. As previously clarified, this scheme perturbs the instruction sequences of a basic block by replacing a statement with one or a sequence of syntactically distinct but semantics-equivalent statements.

As reflected in Table 1, instruction replacement can deceive basic block-level embeddings of other tools. For instance, `ASM2VEC`, which primarily treats instructions as "words" [30], shows a much lower top-1 score (only 19.3 in setting 2) compared with setting 1. However, it is easy to see that the input/output constraints extracted by `sem2vec` are not changed. Therefore, node manipulations would not impede `sem2vec`.

Similarly, another popular obfuscation scheme, often referred to as "garbage code insertion" [52], performs *semantics-preserving* transformations within basic blocks by inserting meaningless instruction sequences without perturbing code semantics. While `sem2vec` is not empirically evaluated against garbage code insertion (`ollvm` does not provide this scheme), we clarify that `sem2vec` shall manifest high resilience toward this scheme due to the usage of symbolic execution and constraint solving.

Opaque Predicate. This scheme (referred to as `-bcf` by `ollvm`) inserts new branches guarded by bogus predicates. Overall, it inserts a tautology path condition hard to analyze; however, this path condition will always be evaluated in one direction ("true" or "false") at runtime. This scheme can generate lots of new blocks and edges. Hence, compared with instruction replacement or garbage code insertion, this scheme more fruitfully complicates the CFG. Although this scheme can introduce relatively high cost, it is shown effective in complicating CFG and impeding binary code matching [52, 90].

`-bcf` effectively undermines all prior works, including both `BinaryAI` and `PalmTree`. As shown in Table 1, the top-1 accuracy of `BinaryAI` on `coreutils` largely drops from 73.9 to 33.7 when using `-bcf`. Similarly, the top-1 accuracy of `PalmTree` drops from 53.7 to 17.3 when using `-bcf`. In contrast, `sem2vec` shows mostly *stable* accuracy when `-bcf` is applied. We find that the garbage code inserted by `-bcf` cannot be optimized out by LLVM optimization passes (even under `-O3`); these garbage codes significantly impede typical binary code embedding tools which are *agnostic* toward program semantics (e.g., deadcode). Given that said, `sem2vec` effectively prunes deadcode introduced by `-bcf` during tracelet-based USE. Hence, `sem2vec` is particularly effective in mitigating this scheme. We view this evaluation (i.e., the *stable accuracy*) as a strong evidence to advocate learning over semantics.

Control Flow Flattening. This scheme "compresses" the CFG into a big "switch" statement. Two dispatcher blocks are deployed to redirect the execution flow while maintaining the original semantics. We looked into the implementation of this scheme in `ollvm` (enabled by `-fla`). Control flow transfers are redirected to the dispatcher blocks inserted by `ollvm`. The dispatcher blocks use a global variable to decide which block to jump next.

Table 2. Vulnerability function search under obfuscation setting `-hybrid`. "1" means an input software with a vulnerability can be matched on correct vulnerability samples in the dataset [26] at top-1. The full output of ASM2VEC is too large to parse; we therefore only report a range ">50".

Vunerability	**CVE**	**Software/Version**	sem2vec$_B$	ASM2VEC	BinaryAI	PalmTree$_B$
Shellshock #1	2014-6271	Bash 4.3	1	1	1	110
		Bash 4.3.30	1	1	1	77
Shellshock #2	2014-7169	Bash 4.3	1	1	4	287
		Bash 4.3.30	1	1	1	238
ffmpeg	2015-6826	ffmpeg 2.6.4	1	1	1	1345
Clobberin´ Time	2014-9295	ntp 4.2.7	1	1	1	95
		ntp 4.2.8	1	1	1	132
Heartbleed	2014-0160	OpenSSL 1.0.1e	1	21	4	1536
		OpenSSL 1.0.1f	1	17	1	1454
		OpenSSL 1.0.1g	1	27	3	2743
wget	2014-4877	wget 1.8	1	1	3	801
ws-snmp	2011-0444	Wireshark 1.12.8	19/5	>50	1	533

sem2vec traverses from the function entry point and always follows the function's normal execution flow, reducing the significant complexity created by `-fla`. Other tools, however, generally treat the entire CFG as a "graph." Thus, a CFG changed by `-fla` is difficult to match with its reference. Given that said, sem2vec is relatively less resistant to `-fla`, because it expands the CFG with more blocks and path constraints. Our manual study shows that sem2vec can pick different input-output constraints for use, thus likely resulting in erroneous matchings. Sec. 8 further discuss methods to reduce erroneous matchings of large CFGs.

Clarification. We discuss sem2vec's resilience toward *commonly-used* obfuscations from both conceptual and empirical perspectives. Under such practical settings, sem2vec delivers highly encouraging results. Nevertheless, sem2vec is **not** designed to be resilient to arbitrary obfuscation schemes. Adversaries may always develop new obfuscations to impede sem2vec (although the cost may be high). Consistent with most, if not all, works, we benchmark sem2vec on *common obfuscations* instead of extreme cases.

6.2 Vulnerability Function Searching

We launch a case study by applying sem2vec to augment a vulnerability search task toward a public vulnerability dataset. This application mimics a common security usage scenario: given an assembly function f from a suspicious piece of executable, we search against a database D of functions with known vulnerabilities and decide if f can be matched with any function in D.

As with ASM2VEC, we use a dataset D released by [26]. This database contains binary code samples of eight CVE vulnerabilities. We evaluate seven CVEs, because the other CVE, `venom`, requires rebuilding `qemu-2.4.0`, which cannot be processed by `ollvm`. D contains 12 assembly functions of seven CVEs (see Table 2), including the infamous Heartbleed exploiting OpenSSL crypto library, and Shellshock allowing remote attackers to execute arbitrary commands on the victim machine. To enhance the difficulty, D also contains 1,225 "negative samples", denoting assembly functions with no vulnerability. A vulnerability search engine must match vulnerable inputs with correct vulnerability samples in D at top-1, without interference from the remainder (benign) functions.

Table 2 compares sem2vec with two other works using the obfuscation setting `-hybrid`. Other obfuscation settings are also evaluated, with similarly promising findings (see below). Overall, this CVE search study reports encouraging results: for 11 out of 12 CVE instances, sem2vec ranks the true match in top-1. When analyzing another infamous CVE, `ws-snmp`, sem2vec achieves a lower accuracy (top-19). We find that this vulnerability contains a large CFG, which hinders sem2vec's tracelet-based SE. To improve sem2vec's comprehension over

Table 3. Vulnerability function search under the obfuscation setting `-sub`.

Vunerability	CVE	Software/Version	sem2vec$_{\mathcal{B}}$	ASM2VEC	BinaryAI	PalmTree$_{\mathcal{B}}$
Shellshock #1	2014-6271	Bash 4.3	1	1	1	1
		Bash 4.3.30	1	1	1	1
Shellshock #2	2014-7169	Bash 4.3	1	1	1	1
		Bash 4.3.30	1	1	1	1
ffmpeg	2015-6826	ffmpeg 2.6.4	1	1	1	1
Clobberin´ Time	2014-9295	ntp 4.2.7	1	1	1	1
		ntp 4.2.8	1	1	1	1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1	1	1	1
		OpenSSL 1.0.1f	1	1	1	1
		OpenSSL 1.0.1g	1	1	1	1
wget	2014-4877	wget 1.8	1	1	1	1
ws-snmp	2011-0444	Wireshark 1.12.8	1	1	1	1

large CFGs, we change the configuration of MAX_STATE from 8 to 16. sem2vec successfully places the true match at top-5 (though spends about 112% more time). BinaryAI extracts some constant strings in its D. We discuss this finding further in Sec. 8.

For three versions of OpenSSL, ASM2VEC and BinaryAI rank the true match much lower. As a result, users of ASM2VEC may need to manually compare at least 17 copies of programs in D to confirm that a Heartbleed vulnerability exists in the suspicious input. PalmTree generally suffers from low accuracy for this evaluation. It places the true matches lower than top-100 or even top-1000 for 10 out of 12 CVE instances. Note that this is consistent with our observation in Table 1; PalmTree generally struggles to match obfuscated code samples, whereas program semantics extracted by sem2vec facilitates much accurate matching in this evaluation.

In addition ot Table 2, which reports evaluation results under obfuscation setting `-hybrid`, we further report evaluation results under obfuscation setting `-sub` (in Table 3), `-bcf` (in Table 4), and `-fla` (in Table 5). In short, sem2vec constantly achieves highly promising results, placing all true matches at top-1 for comparison settings. This illustrates the high robustness of sem2vec's semantics-based approaches. We interpret that sem2vec can effectively augment vulnerability search tasks in real-world scenarios. In contrast, the SOTA model, PalmTree, makes a considerable number of inaccurate matching, particularly for heavily obfuscated binary code like `-fla` (in Table 5). For instance, when enabling control flow flattening (Table 5), the true matching is frequently lower than top-100 to even top-1000. It generally becomes more difficult for users to identify vulnerabilities from the suspicious inputs, given that their true matches are ranked in such low positions.

6.3 Cross-Architecture Evaluation

As mentioned in our application scope discussion in Sec. 4, sem2vec is not limited to x86 platforms. The employed symbolic execution engine, angr, is designed for a multi-platform support. In this section, we assess a challenging comparison setup, cross-architecture binary comparison. In particular, we compile each program in the coreutils dataset into two binaries on the aarch64 architecture and the x86 architecture, respectively. We then cross compare the similarity among each pair of binary code. Table 6 reports the comparison results. We also configure compilers with different optimization levels to enhance the difficulty. Moreover, we underline that the pipeline of sem2vec is trained using only 64-bit x86 binaries. Therefore, sem2vec has no pre-knowledge about the aarch64 platform. Despite this technical challenge, sem2vec manifests a reasonably high accuracy in matching cross-architecture binaries. This illustrates the strength of generating code embedding over symbolic constraints (which is mostly platform independent) rather than the underlying assembly syntax, as most prior binary code embedding works do. We believe the drops in the top-k accuracy as reasonable; the assembly code-level implementation of a function

Table 4. Vulnerability function search under the obfuscation setting `-bcf`.

Vunerability	**CVE**	**Software/Version**	sem2vec$_{\mathcal{B}}$	ASM2VEC	BinaryAI	PalmTree$_{\mathcal{B}}$
Shellshock #1	2014-6271	Bash 4.3	1	1	1	1
		Bash 4.3.30	1	1	1	2
Shellshock #2	2014-7169	Bash 4.3	1	1	1	1
		Bash 4.3.30	1	1	1	17
ffmpeg	2015-6826	ffmpeg 2.6.4	1	1	5	387
Clobberin´ Time	2014-9295	ntp 4.2.7	1	1	1	1
		ntp 4.2.8	1	1	1	1
Heartbleed	2014-0160	OpenSSL 1.0.1e	1	1	1	549
		OpenSSL 1.0.1f	1	1	1	103
		OpenSSL 1.0.1g	1	1	1	65
wget	2014-4877	wget 1.8	1	1	1	67
ws-snmp	2011-0444	Wireshark 1.12.8	1	8	1	33

Table 5. Vulnerability function search under the obfuscation setting `-fla`.

Vunerability	**CVE**	**Software/Version**	sem2vec$_{\mathcal{B}}$	ASM2VEC	BinaryAI	PalmTree$_{\mathcal{B}}$
Shellshock #1	2014-6271	Bash 4.3	1	1	1	198
		Bash 4.3.30	1	1	1	92
Shellshock #2	2014-7169	Bash 4.3	1	1	3	149
		Bash 4.3.30	1	1	4	132
ffmpeg	2015-6826	ffmpeg 2.6.4	1	1	1	1232
Clobberin´ Time	2014-9295	ntp 4.2.7	1	1	1	80
		ntp 4.2.8	1	1	1	54
Heartbleed	2014-0160	OpenSSL 1.0.1e	1	1	1	1430
		OpenSSL 1.0.1f	1	2	1	1413
		OpenSSL 1.0.1g	1	1	1	1425
wget	2014-4877	wget 1.8	1	1	1	944
ws-snmp	2011-0444	Wireshark 1.12.8	1	>50	1	533

Table 6. Top-1/top-3/top-5 accuracy and the corresponding NDCG score for the cross-architecture evaluation using the `coreutils` dataset and sem2vec$_{\mathcal{B}}$.

Comparison			Top-1/3/5	NDCG
`gcc -O0`	vs.	`gcc -O3`	78.5/90.9/93.0	0.892
`gcc -O0`	vs.	`aarch64 -O3`	53.0/68.1/73.2	0.696
`aarch64 -O0`	vs.	`gcc -O3`	46.4/62.9/66.8	0.647
`aarch64 -O0`	vs.	`aarch64 -O3`	68.2/75.2/79.7	0.773

may change accordingly across different architectures. For example, we find that the same function may invoke different library functions when being compiled on different architectures. By comparing setting `gcc -O0` vs. `aarch64 -O3` (3rd row) and `aarch64 -O0` vs. `aarch64 -O3` (5th row), the top-1 accuracy rises significantly when two binaries target the same architecture. Overall, we interepret that `sem2vec` has manifested an encouraging support for cross-architecture binary similarity analysis. It is easy to see that training the underlying models of `sem2vec` with binaries from various architectures would presumably improve its accuracy; we leave it as a future work. Also, since the codebase of `sem2vec` is released [11], audiences can easily re-train `sem2vec` with their own binary samples, as long as they are analyzable by `angr`.

Table 7. Generalizability evaluation using the `OpenSSL` dataset. $\texttt{PalmTree}_G$ denotes `PalmTree` + `Gemini` and $\texttt{sem2vec}_G$ denotes `sem2vec` + `Gemini`. To enhance readablility, for each comparison setting, We mark the better results for each comparison setting **in bold**.

Obf.	**Comparison**	$\texttt{PalmTree}_G$		$\texttt{sem2vec}_G$	
	`gcc -O0` vs.	AUC	Top-1/3/5	AUC	Top-1/3/5
NA	`gcc -O3`	0.897	8.9/15.5/19.3	**0.911**	21.1/29.4/33.8
NA	`clang -O3`	0.891	7.5/12.7/16.4	**0.900**	16.1/23.4/27.1
`-sub`	`ollvm -O3`	**0.887**	7.76/12.6/16.7	0.884	13.7/19.7/22.6
`-bcf`	`ollvm -O3`	0.739	4.1/6.0/7.1	**0.889**	13.9/20.4/24.4
`-fla`	`ollvm -O3`	0.547	2.4/2.8/3.0	**0.858**	10.4/14.8/17.5
`-hybrid`	`ollvm -O3`	0.519	2.2/2.5/2.5	**0.797**	6.6/9.0/10.8

6.4 Generalizability Evaluation

For this evaluation, we aim to benchmark whether `sem2vec` is general enough to be bridged with different graph embedding techniques. To this end, we leverage Gemini [92], a control graph-level embedding framework that is also adopted by `PalmTree`. Therefore, we set up $\texttt{sem2vec}_G$, by using Gemini to encode the tracelet-level embeddings generated by `sem2vec` (implementation details have been discussed before). We reuse the official implementation of `PalmTree` (referred to as $\texttt{PalmTree}_G$), which is based on `Gemini` to compare with $\texttt{sem2vec}_G$. The official implementation ships with two evaluation datasets, `OpenSSL` and `glibc`. We note that `glibc` cannot be compiled by `ollvm`. Therefore, we compare $\texttt{PalmTree}_G$ and $\texttt{sem2vec}_G$ using `OpenSSL` in Table 7 in terms of six comparison settings. Note that $\texttt{PalmTree}_G$ computes AUC scores (higher is better). Therefore, Table 7 reports AUC scores. We also compute top-k scores for each comparison setting as well. Note that `Gemini`'s implementation of AUC computation, to our best understanding, is somehow confusing and potentially buggy.[2] In short, we re-implement a correct version of AUC metrics for $\texttt{PalmTree}_G$ and $\texttt{sem2vec}_G$ when reporting Table 7.

We interpret from Table 7 that $\texttt{sem2vec}_G$ constantly achieves high AUC scores and outperforms the SOTA work, $\texttt{PalmTree}_G$ (except the `-sub` comparison, where $\texttt{sem2vec}_G$ and $\texttt{PalmTree}_G$ have very close AUC scores). Moreover, $\texttt{sem2vec}_G$ manifests higher advantage in terms of the top-k metrics. Overall, AUC scores (used in the `PalmTree` paper) denote a generally more lenient metric than top-k scores. For instance, $\texttt{sem2vec}_G$ achieves about two times higher top-1 scores compared with $\texttt{PalmTree}_G$ for the first three comparison settings. Similar to our observation in Table 1, $\texttt{sem2vec}_G$ shows particularly good accuracy for challenging obfuscated cases, e.g., the top-1 score of $\texttt{sem2vec}_G$ under the `-fla` setting is over four times higher than that of $\texttt{PalmTree}_G$. In short, Table 7 shows that `sem2vec` can deliver effective tracelet-level embedding using different graph-level embedding models and constantly outperform the SOTA work. `sem2vec` is *orthogonal* to particular graph-level embedding models, though $\texttt{sem2vec}_G$ is generally less accurate than $\texttt{sem2vec}_B$.

We also explore if recently released graph neural networks can further improve the accuracy of `sem2vec`. To this end, we replace the backend of `sem2vec`, its adopted GGNN, with a recent graph neural network, Graph Transformer Network (GTN) [96]. GTN is able to generate new graph structures to identify useful relationships between unconnected nodes. It then learns node representations on the new graphs. We adapt the official implementation of GTN to our pipeline, dubbed as $\texttt{sem2vec}_{GTN}$. The model of $\texttt{sem2vec}_{GTN}$ is trained in the same way as the model of $\texttt{sem2vec}_B$, using `coreutils` and `binutils` binary functions compiled with `gcc -O0/O2/O3` and `clang -O0/O2/O3`. Table 8 presents the top-k and NDCG scores of using GGNN and GTN on the dataset of `OpenSSL`, `findutils`, and `libtomcrypt`, in total of 18 comparison settings. $\texttt{sem2vec}_B$ achieves the higher NDCG scores for 12 settings; however, the scores of $\texttt{sem2vec}_{GTN}$ are close. The differences are less than 0.03

[2]See the relevant code snippet at: https://github.com/xiaojunxu/dnn-binary-code-similarity/blob/8552d5b7a35095d901e6e3b3aec62bdc3a1d884e/utils.py#L159

Table 8. Generalizability evaluation using recent advanced graph neural network models. We use the `OpenSSL`, `findutils`, and `libtomcrypt` datasets. `sem2vec`$_B$ denotes `sem2vec` + `BinaryAI` and `sem2vec`$_{GTN}$ denotes `sem2vec` + GTN. To enhance readablility, for each comparison setting, We mark the better results **in bold**.

ID	`OpenSSL` (14160169 × 6 pairs of Function × Function)				
Obf.	**Comparison**	`sem2vec`$_B$		`sem2vec`$_{GTN}$	
	`gcc -O0` vs.	Top-1/3/5	NDCG	Top-1/3/5	NDCG
NA	`gcc -O3`	**61.9**/77.1/81.7	0.770	61.7/78.3/83.0	**0.773**
NA	`clang -O3`	54.1/71.1/77.3	0.738	**64.2**/80.4/85.6	**0.792**
`-sub`	`ollvm -O3`	50.1/65.9/71.9	0.693	47.1/63.4/70.0	0.658
`-bcf`	`ollvm -O3`	**49.7**/66.1/71.7	**0.693**	45.9/62.5/68.8	0.649
`-fla`	`ollvm -O3`	**31.0**/44.5/49.8	**0.506**	28.6/42.2/47.4	0.484
`-hybrid`	`ollvm -O3`	**23.8**/35.1/40.4	**0.434**	19.1/28.3/33.1	0.373
ID	`findutils` (96100 × 6 pairs of Function × Function)				
NA	`gcc -O3`	**72.0**/85.8/89.7	**0.845**	71.4/86.1/89.8	0.842
NA	`clang -O3`	**76.5**/89.0/93.5	**0.863**	71.8/85.3/89.1	0.842
`-sub`	`ollvm -O3`	69.4/86.8/91.9	0.834	**71.1**/83.8/89.5	**0.839**
`-bcf`	`ollvm -O3`	**75.8**/88.1/92.9	0.859	70.7/84.6/88.7	0.837
`-fla`	`ollvm -O3`	61.9/74.5/81.3	0.770	**65.4**/76.7/82.0	**0.788**
`-hybrid`	`ollvm -O3`	**59.0**/75.2/81.0	0.732	53.0/69.2/73.7	0.707
ID	`libtomcrypt` (283024 × 6 pairs of Function × Function)				
NA	`gcc -O3`	**59.9**/77.0/82.2	**.768**	58.6/76.2/80.9	.755
NA	`clang -O3`	59.0/76.7/82.0	.765	**63.6**/80.9/86.2	**.795**
`-sub`	`ollvm -O3`	49.3/67.0/75.5	.688	**53.6**/68.7/78.1	**.717**
`-bcf`	`ollvm -O3`	**56.6**/72.3/78.2	**.740**	50.8/68.7/74.6	.704
`-fla`	`ollvm -O3`	**29.8**/45.1/48.1	**.504**	27.6/40.4/45.1	.482
`-hybrid`	`ollvm -O3`	**26.0**/39.5/47.5	**.475**	19.4/31.7/36.7	.412

Table 9. Top-1/top-3/top-5 accuracy and NDCG evaluation in terms of different settings. To enhance readability, for each comparison setting, we mark the highest top-1 scores and the second-highest top-1 scores .

`diffutils` (25921 × 2 pairs of Function × Function)										
Setting	`sem2vec`$_B$		`PalmTree`$_B$		`BinaryAI`		`asm2vec`		`SAFE`	
`Flatten`	50.3 /69.6/76.4	0.698	31.2/47.7/58.1	0.550	17.7/28.6/36.3	0.407	48.2 /68.7/73.5	0.663	23.2/38.4/46.3	0.478
`Virtualize`	29.8 /47.2/53.4	0.519	4.62/5.38/6.92	0.212	5.14/7.07/8.68	0.210	6.32 /9.89/13.5	0.183	5.18/5.45/6.81	0.196
`zlib` (16900 × 2 pairs of Function × Function)										
`Flatten`	46.2 /60.0/65.4	0.621	34.6 /58.1/69.9	0.614	23.8/45.1/59.8	0.522	1.99/44.4/51.0	0.656	27.7/44.0/56.0	0.525
`Virtualize`	18.5 /30.0/33.8	0.406	1.47/3.68/6.62	0.221	1.64/4.10/5.74	0.213	0.63/4.43/6.96	0.163	2.52 /5.03/6.29	0.214
`gzip` (28224 × 2 pairs of Function × Function)										
`Flatten`	81.5 /88.7/91.7	0.890	13.7/20.9/34.0	0.389	17.6/33.0/39.4	0.430	33.1/51.0/56.1	0.510	33.2 /52.1/58.9	0.572
`Virtualize`	38.7 /49.4/56.5	0.568	8.50/8.50/9.15	0.254	11.7/12.8/13.3	0.279	6.56/7.79/11.9	0.158	13.6 /15.1/16.7	0.283

except the `-hybrid` settings. Therefore, we interpret the usage of GGNN (released at ICLR 2016) in `sem2vec` as sufficient, and replacing it with recent advances in graph neural networks (GTN released at NeurIPS 2019) might not necessarily improve the accuracy much. Rather, according our observation in developing `sem2vec` and experiences in relevant works like BinaryAI [93, 94], the more important factor that influences the quality of binary code embedding would be the input of graph neural networks. Given the same input, different graph neural networks have minor influence on the accuracy, according to our observation and study in this section.

Table 10. Contributions of different semantics features.

S1	**S2**	**S3**	**S4**	**Default**
0.048	0.385	0.552	0.771	0.787

6.4.1 Heavyweight Obfuscation. We have evaluated `sem2vec` using a popular obfuscation, `ollvm`, in Table 1. Overall, while `ollvm` is frequently used in existing works in this field to assess the obfuscation resiliency, we admit that its offered transformation schemes may not be highly intensive. Thus, this section explores using other popular obfuscation schemes to study the resiliency of `sem2vec`. To this end, we employ another commonly-used obfuscator, `Tigress` [17]. `Tigress` is an obfuscation framework over C code. We clarify that this framework provides a number of obfuscation schemes, where most of them appear to have comparable obfuscation strength to `ollvm`. Nevertheless, `Tigress` provides two well-known heavyweight obfuscation methods, control-flow flattening and virtualization-based obfuscation. Our investigation shows that the control-flow flattening scheme of `Tigress` is similar to that of `ollvm`. In contrast, the virtualization-based obfuscation is highly complex. Overall, virtualization-based obfuscation [25] transforms each function into bytecode executing within an interpreter attached in the compiled binary code. Note that the bytecode language can be highly customized and tailored for each individual function. This way, the bytecode is not human readable, and naturally diversified across different functions and different executables. During runtime, each bytecode statement will be interpreted by the attached interpreter to conduct the computation. Virtualization-based obfuscation is generally deemed as one of the most complex obfuscation scheme, which extensively changes the control flow structures and code presentations.

In Table 9, we evaluate those two obfuscation settings offered by `Tigress`; aligned with our previous setup, we also compare the non-obfuscated binary code with its obfuscated version. Here, we compile all binaries with `gcc -O2`. It is worth noting that due to the high complexity of `Tigress`'s setup procedure and its lack of support for the C11 standard, we can only successfully obfuscate and compile `diffutils`(version 3.3), `zlib` (version 1.2.12), and `gzip` [8] (version 1.6). Overall, Table 9 has shown that $\texttt{sem2vec}_B$ can significantly outperform all the other tools for this evaluation. Virtualization-based obfuscation appears to be very effective to undermine the comparison accuracy of all the tools; nevertheless, $\texttt{sem2vec}_B$ still achieves an encouraging accuracy for these settings. Our inspection shows that $\texttt{sem2vec}_B$'s symbolic execution can reasonably track the interpreter's execution, and recover symbolic constraints that are correlated or consistent with constraints obtained when analyzing the non-obfuscated code. Overall, we interpret the obfuscation resiliency of $\texttt{sem2vec}_B$ is promising over different obfuscation schemes. To further improve the accuracy of analyzing virtualization-based obfuscation, we envision the necessity of designing interpreter-aware symbolic execution engines, for instance, jumping over interpreter's routine code during the symbolic execution and only analyzing the semantics of the bytecode. We leave it as one future work to explore.

6.5 Contribution of Semantics-Level Features

Our evaluations show that `sem2vec` manifests promising performance and high robustness, particularly over high-optimized and obfuscated cases. This step aims to understand which semantics-level features primarily contribute to the high quality of computed embeddings. In particular, Considering the extracted semantics-level features in Fig. 3, we set up four settings: 1) **S1** nullifies contribution of semantics but reserves only graph structures, 2) **S2** nullifies contribution of symbolic formulas, 3) **S3** nullifies contribution of external calls, and 4) **S4** nullifies contribution of call stack status (whether the tracelet ends in a callee). We then compare these four settings with the default configuration of `sem2vec`. We re-use the model trained for the experiment of Table 1, and run the evaluation of cross-compiler/optimization using the `coreutils` dataset (i.e., the 1st setting in Table 1). We report the top-1 average accuracy of each setting in Table 10 and compare it with the default setting of `sem2vec`.

Table 11. Top-1 accuracy using different hyper-parameter values. The default setting of these three hyper-parameters are 5, 768, and 8, respectively.

K	1	3	5	7
	0.710	0.750	0.765	0.713
L	192	384	768	1536
	0.723	0.723	0.765	0.643
MAX_STATE	4	8	16	32
	0.720	0.765	0.732	0.680

S1 shows much lower accuracy compared with the other settings, indicating that semantics features notably improve the embedding quality. **S2** implies that symbolic formulas contribute about 40.2% (78.7% - 38.5%) of the overall embedding, whereas **S3** implies that external calls contribute about 23.5% (78.7% - 55.2%). **S4** shows that one-bit call stack status can help our model in several edge cases. Overall, we deduce that features selected by sem2vec are all important to describe the program semantics and facilitate the generation of high-quality embedding.

6.6 Hyper-Parameter Evaluation

sem2vec has three hyper-parameters: 1) K, denoting the extracted $K-1$ input-output constraints (together with one path constraint) from a tracelet's symbolic state (Sec. 4.2), 2) L, representing the size of the function embedding vector (Sec. 4.2), and 3) MAX_STATE, denoting the maximal symbolic states maintained during USE (Sec. 4.1). We assess how different hyper-parameter values can influence sem2vec. For this evaluation, we compute the top-1 accuracy by averaging accuracy scores over all cross-compiler and cross-optimization settings using the coreutils dataset.

Table 11 confirms that when $K = 5$, sem2vec produces embeddings of plausibly better quality. Similarly, $L = 768$ is empirically shown as optimal compared with others. $MAX\_STATE = 8$ achieves the best accuracy, but the results when $MAX\_STATE = 4$ and 16 are also promising. Note that a smaller MAX_STATE can lower the cost of SE, and the case study in Sec. 6.2 also illustrates that a larger MAX_STATE may enhance the accuracy of certain real-world cases with large CFGs. However, setting $MAX\_STATE = 16$ takes approximately 45% more time to finish all SE per function. In short, we interpret that the current hyper-parameter settings in sem2vec are reasonable, and users may fine-tune certain hyper-parameters according to their specific usage scenarios, e.g., increasing MAX_STATE to handle some complex edge cases.

7 DISCUSSION OF EFFECTIVENESS

We present futher discussion, from a qualitative perspective, about the effectiveness of sem2vec and explain why it can outperform other SOTA works, especially when the target binary is obfuscated. Holistically, compared with BinaryAI, sem2vec merely changes the inputs but achieves a much better performance. Therefore, we focus on explaining the quality of collected features of sem2vec, i.e. the inputs of GGNN model, rather than the model itself, its accompanied readout function (i.e., Set2Set), or the training approach (i.e., circle loss). As shown in Fig. 1, the input for the graph embedding model can be separated into the semantics features (e.g., feature vectors of instructions) and structural-level features (e.g., CFG). Thus, we explain the effectiveness of sem2vec from these two aspects.

7.1 Semantic Features

sem2vec relies on the SE engine angr to collect the symbolic constraints first, then uses de facto language embedding model to compute the embedding vectors of those symbolic constraints. To make the SE process scalable, we design our specific traversing algorithm with the under-constrained symbolic execution technique. In addition to scalability, generating high quality constraint embedding is an open problem that is rarely studied in existing works, to the best of our knowledge. Note that the source code is compiled using different optimization and even obfuscation settings, such that the generated constraints "look different," but have identical semantics. For example, an multiply statement in C code may be frequently optimized into a left shifting assembly instruction, as bit shifting is usually rapid on CPUs.

To embed constraints, besides the standard whole word masking task, we design another task, i.e., matching symbolic constraints that come from the same line of source code, as noted in Sec. 4.2. The constraints are collected from binaries compiled with `gcc -O0` and `gcc -O3`. With the debug information, we can link the symbolic constraints and their corresponding source code. From a holistic perspective, this novel pre-training task helps our model learn the knowledge of compiler optimizations, such that the embedding vector of the constraint (derived from extensively optimized code) is compelled to be close to the vector of its un-optimized version. With this improvement, we report that the predicting accuracy increases from 74.5% to 90.1%.

Overall, we admit that some symbolic constraints collected by symbolic execution may look distinct, even if they share identical semantics. From a holistic view, generating closely identical embeddings for symbolic constraints with identical semantics is inherently difficult, because the embedding models do not "understand" the semantics encoded in constraints. In addition to the solution mentioned above, we anticipate that using constraint solvers can help us rule out constraints look similar but indeed not semantics equivalent (and vice versa). That is, conceptually, employing constraint solving can facilitate annotating training data samples in an atuomated manner. We, however, did not take this approach because it is too expensive. As stated in [59], proving the equivalence of two arbitrary symbolic constraints may require iteratively check all permutations of symbolic variables.

Overall, we argue that our current solution, i.e., annotating two symbolic constraints as "equivalent" if they are extracted from the same lines of source code, is a *domain specific solution* with reasonable cost. Though this approach limits the form of semantically equivalent constraints, we find this approach is sufficient in our specific task — binary function similarity analysis — which matches binary functions compiled from the same source code. We leave exploring other methods, e.g., employing constraint solving or data flow analysis, to annotate equivalent/inequivalent constraints as future work. The key challenge is *cost*, given that we requires a considerable amount of annotated symbolic constraints for training.

7.2 Structure-level Features

Besides the embedded vectors for semantics, the quality of structure-level embedding also primiarly influences the accuracy of `sem2vec`. Most binary code embedding works compute structure-level embedding on top of CFG [39], since CFG can be extracted efficiently and precisely with modern reverse engineering frameworks (e.g., IDA Pro). Nevertheless, it is well known that the CFG is "fragile" to compiler optimizations and obfuscation methods (see Fig. 2). The performance of `BinaryAI` and `PalmTree` on obfuscated comparison settings is generally less promising, as shown in Table 1. One important design consideration of `sem2vec` is to avoid the direct usage of naive CFGs. Instead, `sem2vec` traverses the CFG and collects tracelets. This way, `sem2vec` builds a new graph $\mathcal{G}$, where each node in $\mathcal{G}$ is a tracelet (as illustrated in Fig. 3). Symbolic execution on the CFG helps to rule out dead code inserted by obfuscations (Fig. 6.1.2), and therefore, we argue that $\mathcal{G}$ serves as a structural representation of the target function that exhibits much better obfuscation resilience than the original CFG.

For instance, the CFG of `ftp_syst` in `wget` changes significantly after applying the `-hybrid` obfuscation, as shown in Fig. 4(a) and Fig. 4(b). This is reasonable, as `-hybrid` subsumes all three obfuscation passes offered by `ollvm` to transform this function. However, we show that the corresponding "tracelet graph" $\mathcal{G}$ is nearly identical to the original one, as illustrated in Fig. 4(c) and Fig. 4(d). Thus, the generated graph-level embeddings over the tracelet graphs are deemed as resilient to challenging settings like heavyweight optimizations and obfuscations.

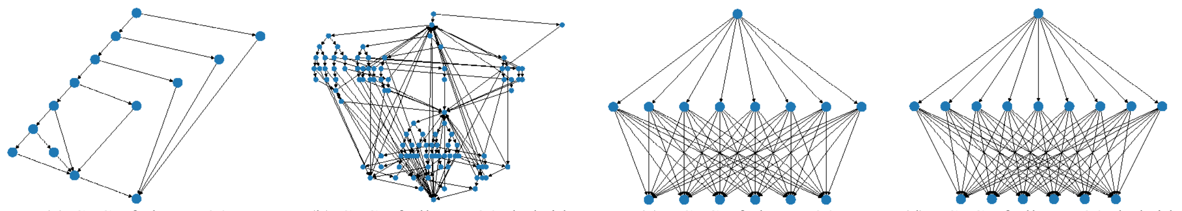

Fig. 4. CFG and tracelet-based CFG (TCFG) of `ftp_syst`. We draw these graphs using the Sygiyama layout algorithm [77]

Table 12. Distribution of FP root causes.

C1	**C2**	**C3**
0.346	0.439	0.215

8 DISCUSSION OF FAILED CASES

This section analyzes false alarms of `sem2vec`. To ease the analysis, we first define **true positive** as we successfully match two assembly functions f and f' compiled from the same source at top-1. Then, false alarms can be classified as follows:

false positive (FP), which implies that assembly functions f and $\hat{f}$ compiled from different source codes are matched at top-1.

false negative (FN), denoting that assembly functions f and f' compiled from the same source code are not matched at top-1.

8.1 False Positives

We investigate 200 randomly-selected FP cases reported in Sec. 6 and summarize the following three FP causes.

C1: f and $\hat{f}$ have almost identical functionality, except some implementation details are changed. For instance, function `rev_xstrcoll_df_atime` and `rev_xstrcoll_df_btime` only differ in calling function `cmp_atime` or `cmp_btime`. Moreover, these two utility functions are mostly identical.

C2: f is matched with $\hat{f}$ due to inlining. In particular, suppose compiler optimization inlines function A into a function B and becomes B'. Then, in our evaluation, a true positive indicates that B' is matched with B, as we rely on the debug symbols to determine if two functions are true match. On the other hand, if B' is matched with A, it is deemed a FP. We agree that it is general obscure to consider "function matching" when inline is taken into consideration. Nevertheless, a closely-related work, `ASM2VEC` [30], uses the same criterion. Besides compiler optimizations (particularly -O3) that actively inline functions, `sem2vec` also inlines callees encountered during its tracelet-based traversal, as clarified in Sec. 4.1. Our manual investigation reveals FP cases due to both compiler optimizations and `sem2vec`'s traversal.

C3: It is not surprising that as long as `sem2vec` cannot match f and f' at top-1, we have one FP. Despite the robustness of semantics learned by `sem2vec`, there are cases where embeddings of f and f' have a longer distance than at least one other pair; see Sec. 8.2.

We interpret FP still matches functions with close functionality in the case of **C1** and **C2**. In other words, code matching failures of these two categories can often provide decent information for real-world applications like code clone detection, vulnerability analysis, and malware clustering. Among 200 analyzed cases, we show the distribution of FP cases in Table 12, where **C1** and **C2** count about 78.5% of "erroneous" matchings. We clarify that the current implementation of `sem2vec` aims to deliver a *general-purpose* framework to match binary code.

Users may extend our released codebase [11] to reduce FP cases. For instance, refraining sem2vec from inlining callee functions to reduce **C2** FPs, though that may potentially undermine identifying true matches in some cases.

8.2 False Negatives

While largely outperforming the state of the arts, sem2vec can still make FNs. Overall, sem2vec uses only K symbolic constraints, external callsites, and call stack to encode the semantics of each tracelet. That is, we sacrifice certain details in code semantics to deliver a practical and efficient framework. Similarly, after manual study, we find cases where our constraint embedding model (Sec. 4.2) treats semantically-equivalent constraints as less "similar" by converting them into embeddings of long cosine distances. This might be due to the inherent limits of neural NLP models. Additionally, sem2vec is trained with only normal code. Obfuscated code (e.g., -fla) that largely complicates the CFG is *not* included in the training dataset of sem2vec. While this design decision is aligned with most prior works in this field [30, 54], we find that most FNs of sem2vec are due to heavily obfuscated cases. From this point, the robustness of sem2vec could be further augmented by *directly training with obfuscated code.*

In Table 2, sem2vec failed on the case ws-snmp by putting its true matching at top-19. Our manual study shows that the employed heavy obfuscation (-hybrid) largely complicates the CFG by adding many extra paths, and each path also becomes more "lengthy." In particular, after the first batch of tracelet-based USE (i.e., executing the ***Traverse_Tracelet*** function in Alg. 1), we find that all collected tracelets have not reached the first call statement in the ws-snmp case. This way, while sem2vec can proceed further and traverse the entire CFG eventually, the induced tracelet graph $\mathcal{G}$ (see Fig. 3) becomes notably different. Tuning the hyper-parameter MAX_STATE from its default value 8 to 16 can resolve this issue, as traversing a tracelet can presumably go further before the maintained symbolic states reach MAX_STATE and terminate this batch of tracelets. We find that the induced tracelet graph $\mathcal{G}$ became succinct after configuring $MAX\_STATE = 16$.

It is also possible to enhance sem2vec by designing strategies to look for more informative constraints. We leave it as one future work to explore using neural attentions [81] to teach sem2vec to prioritize certain constraints. Also, note that BinaryAI has a decent result in Table 2, especially for the ws-snmp case. We clarify that the CVE function has a unique integer constant (0x2DE) and a unique string constant ("pcap"), which are used by BinaryAI as part of features. In contrast, sem2vec normalizes integer constants (Sec. 4.2) and omits strings. sem2vec may incorporate more constants to enhance embedding quality further.

9 RELATED WORK

Sec. 2 has discussed the common pipeline of binary code embedding, and we analyze their common limits in Sec. 3. In this section, we review the techniques of relevant research. A prevalent idea in present binary code embedding works [30, 31, 93, 99] is to apply techniques of natural language processing (NLP) to machine code. ASM2VEC [30], SAFE [61] and DEEPBINDIFF [31] extend the famous word2vec [62] models to produce embedding of instructions. BinaryAI tokenizes instructions and trains binary code embedding models in an end-to-end manner. PalmTree [54] utilizes and augments BERT for instruction-level embeddings. Typically, after computing the embedding vectors of assembly instructions, models like LSTM [42] and HBMP [79] and pooling methods (e.g., average pooling) are used to compute basic block-level embeddings. The next step is to compute embedding vectors for assembly functions, given nearly all binary code-level similarity analysis occurs on the function level. Typically, since basic blocks are not arranged in a linear manner, standard NLP techniques often fail to be applied directly. To solve this challenge, ASM2VEC decomposes the CFG of a function into multiple paths. Since the basic blocks on a path are executed sequentially, standard NLP models can be leveraged smoothly. The state-of-the-art methods, including BinaryAI and PalmTree, use graph neural networks to compute embeddings of function CFG. Nevertheless, as clarified in Sec. 3, the graph structures may be easily changed due to different compilation, optimization, or even obfuscation settings. As described in Table 1, obfuscations applied on the CFG (i.e., -bcf, -fla, and -hybrid) reduce their embedding accuracy. SAFE relies on the attention mechanism to identify the most uncommon sequence of instructions and disregards the structural information. However, Table 1 shows the top-1 and NDCG scores of SAFE are not as good as other works since it utilizes much less information than the other tools.

On the other hand, most conventional techniques leverage program syntactic features for similarity analysis, such as distributions of instructions, opcodes, and system calls [32, 34, 35, 43, 72]. Some works are built based on a "graph view" by extracting control flow and data flow features for comparison [32, 36]. BinDiff, a popular industry tool also evaluated in this works, identifies similar code components through CFG isomorphism comparison [35]. As shown in Sec. 6.1.2, such conventional methods suffer from relatively low performance compared with de facto learning-based methods. Recent studies also point out the key difficulty of extracting proper features w.r.t. diverse sets of binary code samples [51].

Some similarity (and equivalence) analysis is on the basis of symbolic execution [21, 22, 80] and constraint solving. Luo et al. [59] log execution traces during profiling and extract symbolic constraints for comparison. Nevertheless, this work shares a common limitation with prior dynamic methods [23, 44, 47, 48, 65, 73, 84, 86] in terms of low code coverage. Wang et al. [83] blend multiple program execution traces to compute a general and precise program embedding, though it cannot guarantee covering all functions. Another recent work, Trex [67], leverages MLMs to learn from function micro-traces. It alleviates the code coverage issue using transfer learning to generalize knowledge learned over traces.

TRACY [28] decomposes assembly functions also into "tracelets" for comparison. We clarify that "tracelets" are defined in a distinct manner in TRACY and sem2vec: as aforementioned, tracelets in sem2vec denote continuous and short execution traces that are reachable from the function entry point (via symbolic execution). In contrast, tracelets in TRACY denote short sequences of basic blocks on CFGs. Particularly, CFGs are dissected into so-called 3-tracelets (a 3-tracelet contains 3 basic blocks) for matching, despite the fact that some blocks in a tracelet cannot be covered together during runtime, e.g., the third block is a deadcode. This indicates the low resilience of TRACY against optimizations or obfuscations. Two follow-up static works, Esh [26] and GitZ [27], extract strands (i.e., data-flow slices of basic blocks) for comparison. Both methods operate at the boundaries of a basic block. Obfuscation or optimization settings breaking the integrity of basic blocks may likely undermine these two methods. We note that none of these three tools are available for comparison at this point. Moreover, ASM2VEC, which was compared with sem2vec in Sec. 6, has reported to outperform Esh [30] largely.

Code Vectors [40] embeds symbolic execution traces extracted from a function, as opposed to the whole function-level CFG as sem2vec does. The input of Code Vectors is a compilable C project. After analyzing the source code and splitting the program into procedures, Code Vectors performs lightweight SE on each procedure to collect symbolic execution traces. Performing customized, lightweight SE effectively reduces the amount of tokens to be embedded. Since a trace of code vectors is a list of abstract statements, a word2vec model is then trained to embed tokens in each abstract statement by treating a trace as a paragraph. Given that we aim to design sem2vec to be resilient to obfuscation methods, we underline that such meaningful traces in our context could be highly lengthy. Furthermore, the amount of execution traces are often far more than that of the tracelets derived from the CFG, indicating that Code Vectors likely faces the path explosion problem to some degree.

At this step, we compare sem2vec and Code Vectors empirically. Since the "lightweight symbolic engine" of Code Vectors is inapplicable to assembly code, we tentatively implemented the key algorithm of Code Vectors in angr to compare with sem2vec. We unroll loops, skipping to execute callee functions, and omit using constraint solvers. In short, we find that Code Vectors is expensive and less applicable for assembly code. For relatively simple assembly functions with a small number of execution traces, Code Vectors manifests comparable performance with sem2vec. Nevertheless, due to various optimizations like function-inlining, loop unrolling, and tail recursion, assembly functions may be much more complex than their source code versions.

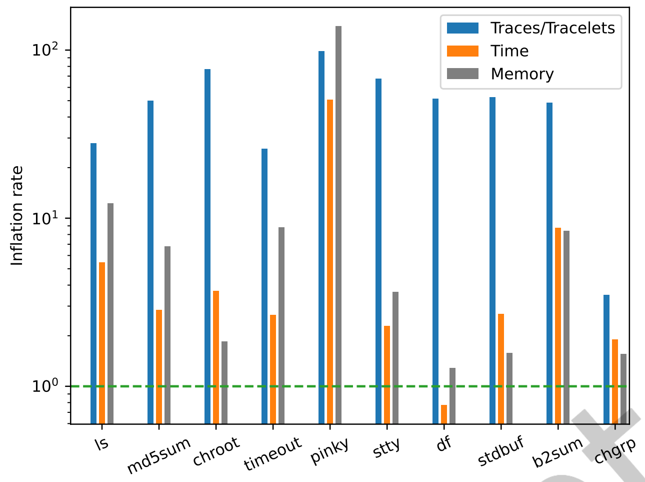

Fig. 5. Performance comparison (inflation rates) between sem2vec and our implemented Code Vectors. We compute the inflation rates in terms of the number of collected traces/tracelets, the processing time, and the memory consumption. All evaluations are launched on the same machine.

Our evaluation uses ten assembly functions with average to relatively large size (over 50 basic blocks) from ten coreutils executables. We report that for an assembly function, sem2vec can collect about 639 tracelets in around 4.5 minute with 3GB of RAM. Note that these 639 tracelets cover the complete CFG of the assembly function. In contrast, our implemented Code Vectors took about 28 minutes and 32GB of RAM to collect on average 33K traces from each assembly function.[3] We find that a significant number of collected traces contain repetitive code blocks. We depict the results breakdown in Fig. 5.

There are also source code embedding works depending on syntax features. Code2vec [15] and Code2seq [14] rely on the abstract syntax tree (AST) of a function to compute embeddings. Overall, they flatten an AST into AST paths, then tokenize the AST paths to compute embeddings. Nevertheless, AST-based methods are not commonly seen in binary code embedding research. We believe the primary reason is because compilers have discarded lots of syntax-level information (e.g., variable types, function names), and the structure of binary code can change significantly due to different compilation configurations (see Fig. 2). CC2Vec [41] learns the representation of software patches without syntax level inputs. Hence, it features the capability of analyzing buggy code that are not even compilable. The input of CC2Vec is the deleted and inserted code lines of a patch, and CC2Vec uses a hierarchical attention network to extract features from the code changes. It is unclear if this hierarchical attention mechanism can manifest high accuracy in computing binary code embedding, given that SAFE, which also features attention, shows less accuracy than other SOTA works.

10 CONCLUSION

We presented sem2vec, a tracelet embedding framework learning over semantics. sem2vec employs both SE and MLM/GNN techniques to achieve a synergistic effect in extracting high-quality and scalable code semantics representations. Our evaluation shows that sem2vec can generate code embeddings that are robust to diverse

[3]We stop collecting traces of a function when we have collected 50K traces.

compilation, optimization, architecture, and obfuscation settings. sem2vec also augments security applications using its quality embeddings.

11 ACKNOWLEDGEMENT

This work was supported in part by CCF-Tencent Open Research Fund. We are grateful to the associate editor and anonymous reviewers for their valuable comments. All correspondence should be addressed to Shuai Wang at the address shown on the first page of this paper.

REFERENCES

[1] 2014. BinDiff. https://www.zynamics.com/bindiff.html.
[2] 2016. RapidJSON. https://rapidjson.org/.
[3] 2022. Binutils. https://www.gnu.org/software/binutils/.
[4] 2022. Coreutils. https://www.gnu.org/software/coreutils/.
[5] 2022. Diffutils. https://www.gnu.org/software/diffutils/.
[6] 2022. Findutils. https://www.gnu.org/software/findutils/.
[7] 2022. GMP. https://gmplib.org/.
[8] 2022. Gzip. https://www.gnu.org/software/gzip/.
[9] 2022. libtomcrypt. https://github.com/libtom/libtomcrypt.
[10] 2022. OpenSSL. https://www.openssl.org/.
[11] 2022. sem2vec Artifact repos. https://github.com/sem2vec.
[12] 2022. zlib. http://zlib.net/.
[13] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. *CoRR* abs/1711.00740 (2017). arXiv:1711.00740
[14] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).
[15] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL (Jan. 2019).
[16] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.
[17] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. 189–200.
[18] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code *(USENIX Security)*.
[19] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics *(NIPS 2018)*.
[20] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. 143–157.
[21] Cristian Cadar. 2015. Targeted program transformations for symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 906–909.
[22] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8.
[23] Silvio Cesare and Xiang Yang. 2012. *Software Similarity and Classification*. Springer Science.
[24] 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)*.
[25] Christian Collberg. 2021. The Tigress C Diversifier/Obfuscator – Virtualization. http://tigress.cs.arizona.edu/transformPage/docs/virtualize/index.html.
[26] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries *(PLDI)*.
[27] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 79–94.
[28] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, 349–360.
[29] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).

[30] S. H. Ding, B. M. Fung, and P. Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *IEEE S&P*.

[31] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DEEPBINDIFF: Learning Program-Wide Code Representations for Binary Diffing. (2020).

[32] Thomas Dullien and Rolf Rolles. 2005. Graph-based comparison of executable objects. *SSTIC* (2005).

[33] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* (2008).

[34] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code *(NDSS)*.

[35] Halvar Flake. 2004. Structural Comparison of Executable Objects *(DIMVA)*.

[36] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable Detection of Semantic Clones. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*. ACM, 321–330.

[37] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs *(ICICS)*.

[38] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *ICML*.

[39] Irfan Ul Haq and Juan Caballero. 2019. A Survey of Binary Code Similarity. *arXiv preprint arXiv:1909.11424* (2019).

[40] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. 2018. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 163–174.

[41] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 518–529.

[42] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[43] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale Malware Indexing Using Function-call Graphs *(CCS)*.

[44] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2016. Cross-Architecture Binary Semantics Understanding via Similar Code Comparison *(SANER)*.

[45] Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference. In *USENIX Security*.

[46] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[47] Y. C. Jhi, X. Jia, X. Wang, S. Zhu, P. Liu, and D. Wu. 2015. Program Characterization Using Runtime Values and Its Application to Software Plagiarism Detection. *IEEE Transactions on Software Engineering* (2015).

[48] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. 2011. Value-based Program Characterization and Its Application to Software Plagiarism Detection *(ICSE)*.

[49] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. 2012. Binary Function Clustering Using Semantic Hashes *(ICMLA)*.

[50] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM: Software Protection for the Masses *(SPRO)*.

[51] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2020. Revisiting Binary Code Similarity Analysis using Interpretable Feature Engineering and Lessons Learned. *arXiv preprint arXiv:2011.10749* (2020).

[52] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE S&P*.

[53] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. PMLR, 1188–1196.

[54] Xuezixiang Li, Qu Yu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. (2021).

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

[56] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[57] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[58] Sifei Luan, Di Yang, Koushik Sen, and Satish Chandra. 2018. Aroma: Code Recommendation via Structural Code Search. *CoRR* abs/1812.01158 (2018). arXiv:1812.01158 http://arxiv.org/abs/1812.01158

[59] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *FSE*.

[60] 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 Trans. Softw. Eng.* 43, 12 (Dec. 2017), 1157–1177.

[61] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proceedings of 16th Conference on Detection of Intrusions and Malware Vulnerability Assessment*

(DIMVA).

[62] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[63] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking *(USENIX)*.

[64] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver *(TACAS)*.

[65] Ginger Myles and Christian Collberg. 2004. Detecting Software Theft via Whole Program Path Birthmarks *(ISC)*.

[66] Feiping Nie, Wei Zhu, and Xuelong Li. 2017. Unsupervised large graph embedding. In *Thirty-first AAAI conference on artificial intelligence*.

[67] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv preprint arXiv:2012.08680* (2020).

[68] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic comparison of symbolic execution systems: intermediate representation and its generation. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 163–176.

[69] David A. Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code *(USENIX)*.

[70] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[71] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 142–157.

[72] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting Code Clones in Binary Executables *(ISSTA)*.

[73] David Schuler, Valentin Dallmeier, and Christian Lindig. 2007. A Dynamic Birthmark for Java *(ASE)*.

[74] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.

[75] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE S&P*.

[76] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Baishakhi Ray. 2018. Obfuscation resilient search through executable classification. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 20–30.

[77] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. 1981. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics* 11, 2 (1981), 109–125.

[78] Yifan Sun, Changmao Cheng, Yuhan Zhang, Chi Zhang, Liang Zheng, Zhongdao Wang, and Yichen Wei. 2020. Circle loss: A unified perspective of pair similarity optimization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 6398–6407.

[79] Aarne Talman, Anssi Yli-Jyrä, and Jörg Tiedemann. 2019. Sentence embeddings in NLI with iterative refinement encoders. *Natural Language Engineering* 25, 4 (2019), 467–482.

[80] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped symbolic execution. In *Proceedings of the 40th International Conference on Software Engineering*. 350–360.

[81] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[82] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391* (2015).

[83] Ke Wang and Zhendong Su. 2020. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 121–134.

[84] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *ASE*.

[85] Wendong Wang, Joshua Giltinan, Svetlana Zakharchenko, and Metin Sitti. 2017. Dynamic and programmable self-assembly of micro-rafts at the air-water interface. *Science Advances* 3, 5 (May 2017), e1602522. https://doi.org/10.1126/sciadv.1602522

[86] X. Wang, Y. C. Jhi, S. Zhu, and P. Liu. 2009. Detecting Software Theft via System Call Based Birthmarks. In *ACSAC*.

[87] Zhihao Wang, Jian Chen, and Steven C. H. Hoi. 2019. Deep Learning for Image Super-resolution: A Survey. http://arxiv.org/abs/1902.06068 cite arxiv:1902.06068.

[88] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[89] Zhang Xinyi and Lihui Chen. 2018. Capsule graph neural network. In *International conference on learning representations*.

[90] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 921–937.

[91] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[92] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *CCS*.

[93] 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. (2020).

[94] Zeping Yu, Wenxin Zheng, Jiaqi Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2020. CodeCMR: Cross-Modal Retrieval For Function-Level Binary Source Code Matching. *Advances in Neural Information Processing Systems* 33 (2020).

[95] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security*.

[96] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph transformer networks. *Advances in neural information processing systems* 32 (2019).

[97] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 793–803.

[98] Fangfang Zhang, Yoon-Chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. 2012. A First Step Towards Algorithm Plagiarism Detection. In *ISSTA*.

[99] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2019. Neural Machine Translation Inspired Binary Code Similarity Comparison beyond Function Pairs. In *NDSS*.