Semantic Learning and Emulation Based Cross-platform Binary Vulnerability Seeker

Jian Gao, Yu Jiang, Zhe Liu, Xin Yang, Cong Wang, Xun Jiao, Zijiang Yang, Jiaguang Sun

Abstract—Clone detection is widely exploited for software vulnerability search. The approaches based on source code analysis cannot be applied to binary clone detection because the same source code can produce significantly different binaries due to different operating systems, microprocessor architectures and compilers. In this paper, we present *BinSeeker*, a cross-platform binary seeker that integrates semantic learning and emulation. With the help of the labeled semantic flow graph, *BinSeeker* can quickly identify M candidate functions that are most similar to the vulnerability from the target binary. The value of M is relatively large so this semantic learning procedure essentially eliminates those functions that are very unlikely to have the vulnerability. Then, semantic emulation is conducted on these M candidates to obtain their dynamic signature sequences. By comparing signature sequences, *BinSeeker* produces top-N functions that exhibit most similar behavior to that of the vulnerability. With fast filtering of semantic learning and accurate comparison of semantic emulation, *BinSeeker* seeks vulnerability precisely with little overhead. The experiments on six widely used programs with fifteen known CVE vulnerabilities demonstrate that *BinSeeker* outperforms three state-of-the-art tools *Genius*, *Gemini* and *CACompare*. Regarding search accuracy, *BinSeeker* achieves an MRR value of 0.65 in the target programs, whereas the MRR values by *Genius*, *Gemini* and *CACompare* are 0.17, 0.07 and 0.42, respectively. If we consider ranking a function with the targeted vulnerability in the top-5 as accurate, *BinSeeker* achieves the accuracy of 93.33%, while the accuracy of the other three tools is merely 33.33%, 13.33% and 53.33%, respectively. Such accuracy is achieved with 0.27s on average to determine whether the target binary function contains a known vulnerability, and the time for the other three tools are 1.57s, 0.15s and 0.98s, respectively. Compared to the time used to manually identify the true positive vulnerability from the false positive candidates reported by Gemini, the time overhead of *BinSeeker* is negligible. Evidently, the proposed *BinSeeker* achieves a better balance between accuracy and efficiency.

Index Terms—semantic emulation, semantic learning, cross-platform binary, vulnerability search, neural network.

1 Introduction

Pre-existing code has been widely reused to improve software development productivity. Releases of different software products contain significant amounts of identical or similar code, a phenomenon known as code clone. The related study has shown that 22.3% of the Linux kernel is from the previous implementation [1]. Consequently, along with the widespread usage of code clone, vulnerabilities spread as well because the code fragments containing the vulnerabilities are reused in other software products. In addition, the patches that fix vulnerabilities are not automatically propagated, thus the cloned vulnerabilities are not patched even though the original code has been fixed. For example, 145 unpatched cloned vulnerabilities were confirmed in the Debian system [2].

In order to maintain code quality, it is critical to identify cloned code once a vulnerability is detected. Unfortunately, it is a very challenging task because the same source code can be compiled with different compilation parameters and even different compilers. In addition, with the popularity of terminal devices, software programs on traditional *X86* architecture are gradually ported to other architectures such as *ARM* and *MIPS*. These different environments produce different binaries given the same source code. To address this challenge, cross-platform binary vulnerability search has attracted increasing attention in recent years, and three types of approaches have been proposed: static, dynamic and learning approaches [3], [4], [5], [6], [7], [8], [9].

Static approaches usually rely on the graph matching algorithm on control flow graphs (CFGs) to identify binary code similar to the vulnerable code [3], [4], [10], [11]. However, CFGs of the same function differ significantly with different compilation configurations (e.g., *O0–O3* optimization levels, *X86* and *MIPS* processor architectures), which leads to inaccurate results. In order to enhance accuracy, Pewny *et al.* [11] presents a method to broaden the analysis from the similarity of basic blocks to the similarity of functions based on CFGs. Although this method improves accuracy, having too many false positives prevents it from being widely adopted.

Dynamic approaches overcome the obstacle of inaccurate search through monitoring the runtime traces of binary programs in real operating environments and then performing equivalence checking between two traces [5], [6]. However, significant overhead is inherent and dynamic approaches are often expensive in practice. For example, although *CACompare* [6] achieves over 80% search accuracy, it takes *CACompare* about three hours to generate function signatures and about seven hours to complete the search task for the *OpenSSL* with 5,995 functions.

As a burgeoning technique, learning-based approaches are increasingly applied to binary vulnerability search because of fewer domain knowledge requirements [7], [8], [9]. Since most of these approaches rely only on the CFGs to convert assembly instruction features into numerical vectors, they can quickly predict whether a binary function is vulnerable. Unfortunately, their accuracy is rather low. For example, the top-50 accuracy of *Genius* for two case studies on a large set of firmware images is only 28% and 48% [7]. With such a low accuracy, nontrivial manual effort is often required to eliminate false positives, which is not practical for industrial applications. To enhance the robustness against structural differences in the CFG, we propose *VulSeeker* [9], a semantic learning-based approach that adopts lightweight instruction features and integrates

DFG into CFG. Furthermore, we also attempt to supplement multiple semantic signatures in *VulSeeker-Pro* [12] to evaluate function similarity against compilation optimization differences on the same architecture.

Both learning-based and dynamic approaches have barriers to use when it comes to large-scale code. For example, consider such a scenario to determine whether a large number of cross-architecture embedded firmware contain a serious vulnerable function that has just been exposed. In this case, applying the learning-based approach alone can quickly predict the similarity between any function within the firmware and the vulnerable function. Since it only uses vectors to represent the cross-architecture function semantics which can be insufficient, additional human efforts are required to participate in the confirmation process. On the other hand, the dynamic approach can well match the same vulnerable function of different platforms, but may suffer from time-demand bottlenecks of large-scale code. Execution for all functions is extremely time consuming.

To overcome the above limitations and strike a balance between accuracy and efficiency, we present *BinSeeker*[1], a cross-platform binary vulnerability seeker that integrates learning and dynamic approaches. It seamlessly integrates our previous works [9], [12] and supports the cross-architecture function emulation to achieve better performance. The intuitive idea is to apply the dynamic approach to the quickly acquired narrow range of candidate functions that are similar to the vulnerability, which can both improve the search accuracy and reduce the time requirement. Similar idea has also been applied in Driller for vulnerability fuzzing, that combine the cheap searched based AFL to achieve fast coverage and the expensive symbolic execution to solve the constraints hard to be searched.

Given a vulnerable function, the learning component in *BinSeeker* quickly identifies M candidate functions from the target binary that are most similar to the vulnerable function. With the aid of the labeled semantic flow graph, *BinSeeker* captures more function-level semantics and is more accurate than the existing learning-based approach [8]. However, M has to be relatively large due to the inherent inaccuracy in all learning-based approaches; otherwise, false negatives are very likely to occur. To avoid manual efforts of examining all M functions, the semantic emulation component of *BinSeeker* conducts dynamic emulation on the M candidates and identifies top-N functions that are most similar to the vulnerable function. That is, *BinSeeker* first conducts a computationally cheap and less accurate search on the entire target program using semantic learning, and then performs an expensive and more accurate search on the M candidate functions using semantic emulation. With N being much smaller than M, *BinSeeker* obtains highly accurate results at a cost almost the same as that of a semantic learning-based approach.

To evaluate *BinSeeker* on widely-used open-source applications such as *OpenSSL*, we compare it with three state-of-the-art cross-platform binary vulnerability search tools: *Genius* [7] and *Gemini* [8] which are semantic learning-based tools and *CACompare* [6] which is a semantic emulation-based tool. The experimental results show that *BinSeeker* significantly outperforms the three tools in comparison. Regarding vulnerability search accuracy, *BinSeeker* achieves an MRR (mean reciprocal rank) value of 0.65 in the target programs, whereas the MRR values of *Genius*, *Gemini* and *CACompare* are 0.17, 0.07 and 0.42, respectively. If we consider ranking a function with the targeted vulnerability in the top-5 as accurate, *BinSeeker* achieves a top-5 accuracy of 93.33%, while the top-5 accuracy of the other three tools is merely 33.33%, 13.33% and 53.33%, respectively. In terms of time used in the search, it takes *Genius*, *Gemini*, *CACompare* and *BinSeeker* 8,992s, 849s, 8,432s and 1,323s respectively to complete a search task on the *OpenSSL* binary. The results demonstrate that *BinSeeker* is more accurate than semantic emulation-based tools and the cost is comparable to that of the learning-based approach. These advantages reduce the burden of engineers when manually identifying true positive cases. In summary, the present study makes the following contributions:

- To date, *BinSeeker* is the first tool that integrates semantic learning and emulation to improve search accuracy and efficiency. Accuracy is critical as it reduces manual efforts to inspect a large number of functions. Meanwhile, *BinSeeker* achieves highly accurate results at a cost almost the same as that of the fast semantic learning-based approach.
- We optimize the original learning and emulation approaches to improve *BinSeeker*'s performance. Consequently, the learning component is more accurate with the extended labeled semantic flow graph and the emulation engine is more lightweight and faster with the optimized function signature extraction.

The rest of this paper is organized as follows: Section 2 introduces the background to help understand our neural network model; Section 3 details the design and implementation of *BinSeeker*; Section 4 describes experimental results compared to the state-of-the-art approaches; and Section 5 discusses challenges and future work. Section 6 delineates related work and Section 7 presents the conclusion.

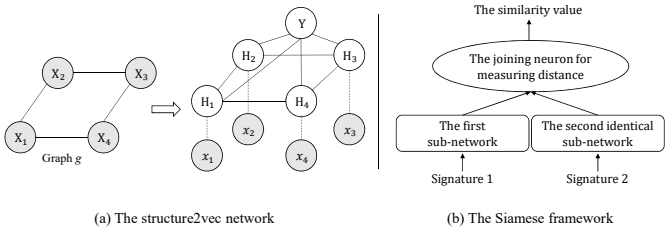
2 Background

The semantic learning module of *BinSeeker* is based on the *Siamese* framework [13] and the *structure2vec* network [14], as shown in Fig. 1.

a) Application of the *Siamese* Framework and the *structure2vec* Network to Vulnerability Search. In order to identify the functions that are most similar to the one with known vulnerability, a key step is to compute the similarity between a pair of functions. In *BinSeeker*, the input pair of functions are represented as LSFGs (described in Section 3.1.1), and the neural networks are dedicated to the LSFG structures. The embedding vector generation strategy of the *structure2vec* network transforms the vulnerability search problem to be the problem of calculating the similarity between function embedding vectors. Given two graphs g_1 and g_2, the label y in the training tuple $\langle g_1, g_2, y\rangle$ indicates whether the two graphs are similar. This dual-input and single-output tuple requires us to train two *structure2vec* networks, one for each graph, with shared parameters. Thus, we embed two identical *structure2vec* networks into the *Siamese* framework shown in Fig. 4(a), which is described in detail in Section 3.1.3. The final effect is that the vector representations of similar graphs are close to each other.

b) The *structure2vec* Network. Dai *et al.* [14] proposed the *structure2vec* graph neural network, which proved to be

1. The prototype implementation is open-source and is available at



(a) The structure2vec network (b) The Siamese framework

Fig. 1. Two foundational network structures used in *BinSeeker*.

effective and scalable for structured data representation. The vertices in the graph are connected to each other by the edges. The *structure2vec* network can encode vertex features and connection relationship of the edges in the graph as the embedding vector to represent graph semantics. Fig. 1(a) shows how the *structure2vec* network works to generate the embedding vector based on the graph g on the left. The example graph contains four vertices X_i, $i \in \{1, 2, 3, 4\}$, and each vertex is appended with an initial numerical vector x_i. The right side of Fig. 1(a) is the *structure2vec* network structure that consists of one input layer, T hidden layers (only one hidden layer is drawn for illustration), and one output layer. In the *structure2vec* network, the number of neuron nodes in the input layer and each hidden layer is the number of vertices in the original graph g.

In the input layer, the input of each neuron node is the initial numerical vector x_i of the corresponding vertex. In the t^{th} ($1 \leq t \leq T$) hidden layer, each hidden neuron H_i is responsible for generating a new feature representation called the embedding vector $\widetilde{\mu}_i^{(t)}$. We use formula $\widetilde{\mu}_i^{(t)} = F(x_i, \Sigma_{k \in E(i)} \widetilde{\mu}_k^{(t-1)})$ to represent the mapping relationship of each hidden neuron, where $E(i)$ refers to the set of vertices adjacent to vertex X_i. Through the mapping function F, the feature of each vertex is propagated to other vertices based on connected edges. This feature update strategy takes into account the graph topology and ensures that each vertex incorporates information from neighbors within T hops after T hidden-layer iterations. Finally, the output layer neuron Y aggregates the output embedding vectors $\widetilde{\mu}_i^{(T)}$ of the T^{th} hidden layer neurons to form the final embedding vector of the entire graph.

c) The *Siamese* Framework. The *Siamese* framework [13] shown in Fig. 1(b) solves the dual input problem of verification of signatures written on a pen-input tablet. It mainly consists of two identical sub-networks, each of which takes a processed signature as its input and then outputs the feature of the signature. The joining neuron is used to measure the distance between these two output features of the two sub-networks and output the similarity value ranging from -1 to 1. In addition, it is used as a network framework to train two identical embedded sub-networks in an end-to-end manner, which ensures that the two sub-networks share the same parameters.

3 Design of *BinSeeker*

Our objective is to automatically identify whether a given binary program contains known vulnerabilities or not. Lots of vulnerability-related databases are open to the public. For example, *Common Vulnerabilities and Exposures* (CVE) [15] contains more than 120,000 vulnerabilities. In the case of open-source software, which is often reused to improve software development, some of the CVEs provide an explicit indication of the source functions involved in the vulnerabilities. Based on the indication of these known vulnerabilities within a single function, clone-based search approaches at the function-level granularity will help to detect those known vulnerabilities in target cross-platform binaries.

As presented in Fig. 2, *BinSeeker* contains two major modules: *semantic learning* and *semantic emulation*. Its inputs are a specific vulnerability and a target binary to be searched. *BinSeeker* utilizes the fast predictive capability of the customized deep neural network to obtain the initial M (e.g., 200) candidate functions by eliminating extremely dissimilar functions in the *semantic learning* module. Then, *BinSeeker* sorts the M candidate functions based on function dynamic signature sequences to generate top-N (e.g., top-25) candidate functions as the final prediction results for the vulnerability in the *semantic emulation* module.

3.1 Semantic Learning

The main goal of this module is to quickly eliminate remarkably dissimilar functions from the target binary and get the top-M candidate functions that are most similar to a specific vulnerability. Its pivotal capability is to obtain embedding vectors representing the function semantics and use them for similarity comparison according to three steps: LSFG construction, feature extraction, and customized deep neural network.

3.1.1 LSFG Construction

Many existing methods rely on CFGs as a basis to obtain function semantic representations. These representations are either the symbolic formulas encoding the input-output relationship of basic blocks [10], [11], [16] or the attributed control flow graphs containing instruction features [7], [8]. Function semantics obtained through these methods can be highly inaccurate since the CFGs show significant diversities under different compilation scenarios [17].

We propose the labeled semantic flow graph (LSFG) which combines the CFG and the data flow graph (DFG) to capture more accurate function semantics. The idea is based on the fact that the CFG determines the possible execution sequences of basic blocks and the DFG depicts the transfer and use of data within the function. The combination of these two dependent relations makes function semantics (explained in Section 3.1.3) resistant to structural and syntactic differences in the CFGs under different architectures and compilation optimization strategies.

LSFG is different from the program dependence graph (PDG) [18] and hybrid information- and control-flow graph (HI-CFG) [19] which also consists of DFG and CFG. They work at different code granularity as well. For example, PDG establishes an edge connection on the statement or instruction granularity, while LSFG is on the basic block granularity. HI-CFG not only needs to create data structure nodes but also code block nodes. It also needs to use the trace-based dynamic analysis approach to infer the edge connections between these two types of nodes. Considering that the complexity of the HI-CFG and PDG structures sharply increases the processing time of the semantic learning model, we choose the more lightweight LSFG graph representation which only creates code block nodes in the

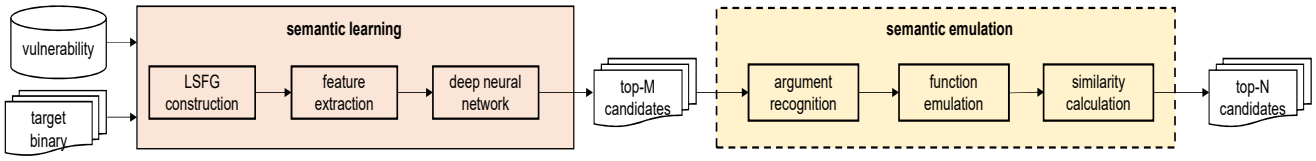
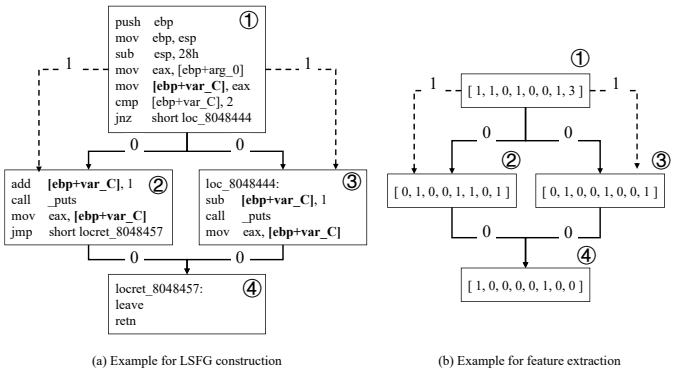


Fig. 2. Overall workflow of *BinSeeker*: the first phase relies on semantic learning to quickly predict top-M most similar candidate functions with a low time cost, the second phase employs function emulation to output more accurate top-N candidates, where M and N can be dynamically configured.

CFG but ignores the data dependence edges within each basic block of the CFG. In terms of the efficiency and accuracy of the experiment, the proposed LSFG is suitable for the customized structure2vec-based semantic learning network to learn the numerical semantic representation of the binary function.



(a) Example for LSFG construction (b) Example for feature extraction

Fig. 3. An example for the deep neural network input vector generation.

Fig. 3(a) illustrates an example of the proposed LSFG. The solid lines (labeled by 0) represent control dependency, while the dotted lines (labeled by 1) represent data dependency. For a compiled binary function, the structure of its assembly function is organized according to the CFG, which can be easily parsed with the help of common disassembly software (e.g., *IDA Pro* [20], *angr* [21]). CFGs obtained by different methods are almost the same, while DFGs differ according to different data dependence rules. In the present paper, we construct the DFGs on top of CFGs by leveraging the *define-use* rules and traversing all function paths. Specifically, for two instructions i and j from two different basic blocks, if the instruction i writes to a memory location and the instruction j reads from the same memory location, we create a data dependence edge from the basic block of i to the basic block of j. It is worth noting that there is at most one data dependence edge between two different basic blocks. When a variable is directly read without being written in the current basic block, we look for the pre-ordered basic block in which the variable is last defined, and then establish the data pointing edge between the two basic blocks unless the variable is never defined in the function. Therefore, the presence of memory address "$[ebp + var\_C]$" forms a data dependence edge between the block ① and ② in Fig. 3(a).

3.1.2 Feature Extraction

The LSFG constructed above is not suitable yet as input into our customized neural network (NN) model. The purpose of feature extraction is to generate the block-level initial numerical vectors related to functions that can be input into the NN model to generate function-level embedding vectors for similarity calculation. We should choose and extract robust and lightweight features that change little under various implementation platforms with different microprocessor architectures and various compilation optimization configurations as initial numerical vectors. By empirically referring to features used in previous works [3], [7], [22] and executing a series of experiments (described in Section 4.2.3) for different feature sets, we finally determine to use the 8 types of features shown in Table 1.

TABLE 1
Basic-block level features used by *BinSeeker*.

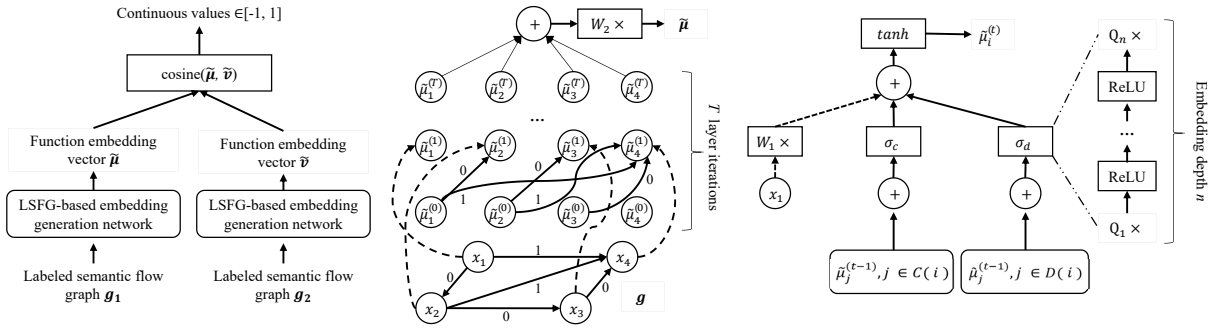
Feature Name	Example
No. of stack operation instructions	push, pop
No. of arithmetic instructions	add, sub
No. of logical instructions	and, or
No. of comparative instructions	test
No. of library function calls	call printf
No. of unconditional jump instructions	jmp
No. of conditional jump instructions	jne, jb
No. of generic instructions	mov, lea

We first count the number of each feature in each basic block, then arrange them into numerical vectors in order, and finally put these numerical vectors at the corresponding vertex of LSFG. Fig. 3(b) presents the numerical vectors of each basic block corresponding to the function in Fig. 3(a). We denote the LSFG with numerical vectors as $g = \langle X, C, D \rangle$, where X, C and D are the sets of (basic block) vertices, control dependence edges and data dependence edges, respectively. Each vertex $x_i \in X$ represents the initial numerical feature vector. The LSFG mentioned later in this paper refers to the LSFG with initial numerical feature vectors unless otherwise specified. Paired LSFGs g_1 and g_2 are the input of the *BinSeeker* neural network.

3.1.3 Customized Deep Neural Network

Because the two basic networks introduced in Section 2 satisfy our vulnerability search requirement, this part clarifies how to combine the *Siamese* [13] framework with the adapted *structure2vec* [14] network to implement our customized network. Since the *structure2vec* network is implemented to support LSFGs, we call it the LSFG-based embedding generation network.

a) *BinSeeker* Network Structure. Fig. 4(a) shows the overall architecture of the *BinSeeker* neural network model. Its inputs are two LSFGs, which are abbreviated as g_1 and g_2. These two graphs (g_1 and g_2) are imported into two identical LSFG-based embedding generation networks, which



(a) The network model of BinSeeker for vulnerability search

(b) LSFG-based embedding generation network

(c) A schematic diagram for generating the embedding vector of each vertex during t^{th} iteration.

Fig. 4. The network of *BinSeeker* for vulnerability search, including the customized similarity calculation, embedding and vector generation.

transform the structured graph information into function-level embedding vectors (e.g., $\widetilde{\mu}$, $\widetilde{\nu}$) capturing the function semantics. The *Cosine* function is used to calculate the similarity of two embedding vectors, which represents the similarity of two binary functions.

Fig. 4(b) is a detailed description of the LSFG-based embedding generation network. This process is similar to the basic *structure2vec* except for the graph type. We extend *structure2vec* to deal with LSFG containing both data flow (labeled 1 on the solid arrow) and control flow (labeled 0 on the solid arrow), while basic *structure2vec* works with an ordinary undirected graph. In the input layer, the input example LSFG $g = \langle X, C, D \rangle$ consists of four vertices X_i, $i \in \{1, 2, 3, 4\}$, each of which represents a basic block of the function and contains the block-level initial feature vector x_i. $C(i)$ and $D(i)$ represent the control dependence edge set and the data dependence edge set of vertex i, respectively.

The adapted *structure2vec* network contains a total of T hidden layer responsible for transforming graph information into the function semantic embedding vector. Each hidden layer node is represented as the updated block-level embedding vector $\widetilde{\mu}_i^{(t)}$, where different values of t correspond to different hidden layers. During the t^{th} hidden layer iteration, the updated $\widetilde{\mu}_i^{(t)}$ consists of three different inputs: the initial feature vector x_i of the corresponding vertex X_i (the dotted arrow in Fig. 4), the sum $l_c^{(t-1)} = \Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}$ of previous embedding vectors of vertices pointing to X_i through the control dependency $C(i)$, and the sum $l_d^{(t-1)} = \Sigma_{j \in D(i)} \widetilde{\mu}_j^{(t-1)}$ of previous embedding vectors of vertices pointing to X_i through the data dependency $D(i)$. The updated $\widetilde{\mu}_i^t$ is expressed by mapping function F with the formula: $\widetilde{\mu}_i^{(t)} = F(x_i, l_c^{(t-1)}, l_d^{(t-1)}) = tanh(\, W_1 x_i + \sigma_c(l_c^{(t-1)}) + \sigma_d(l_d^{(t-1)}))$. Fig. 4(c) illustrates the procedure for generating the embedding vector $\widetilde{\mu}_i^{(t)}$ of each hidden node, where σ_c, σ_d are two non-linear transformation functions which are responsible for calculating an embedding vector with more powerful representation capability. Similar to [8], we define them as two n layer fully-connected networks with the following equations:

$$\begin{cases} \sigma_c(l_c) = P_1 \times ReLU(P_2 \times \cdots ReLU(P_n \times l_c)) \\ \sigma_d(l_d) = Q_1 \times ReLU(Q_2 \times \cdots ReLU(Q_n \times l_d)) \end{cases}$$

where n is the embedding depth representing the number of layers in a fully-connected network, P_i and Q_i are $p \times p$ dimensional parameter matrices for each layer of the two fully-connected networks.

The overall procedure for generating function semantic representation described above is integrated into the Algorithm 1. W_1 and W_2 are $d \times p$ and $p \times p$ dimensional parameter matrices, respectively. Through the T-layer iteration from Lines 5 to 11 in Algorithm 1, a new embedding vector of each vertex is generated, which not only follows the topology structure of LSFG but also integrates the T-hop interaction among vertices. In other words, the features of the vertices are propagated to other vertices as the iteration progresses, ensuring that each basic block within the function incorporates information from neighbors within T hops after T hidden-layer iterations. Finally, the binary function semantics, including the data flow dependency and the control flow dependency, is aggregated into the function-level embedding vector $\widetilde{\mu}$ in Line 12.

Algorithm 1: Generating function semantics

Input: LSFG $g = \langle X, C, D \rangle$
Hidden layer iteration number T
Output: Binary function semantics embedding vector $\widetilde{\mu}$

1 $C(i)$ as the set of parent nodes that are the control dependency of vertex i; $D(i)$ as the set of parent nodes that are the data dependency of vertex i.
2 **for** $i \in X$ **do**
3 $\quad \widetilde{\mu}_i^{(0)} = 0$
4 **end**
5 **for** $t = 1$ *to* T **do**
6 $\quad$ **for** $i \in X$ **do**
7 $\quad\quad l_c^{t-1} = \Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}$
8 $\quad\quad l_d^{t-1} = \Sigma_{j \in D(i)} \widetilde{\mu}_j^{(t-1)}$
9 $\quad\quad \widetilde{\mu}_i{}^{(t)} = tanh(\, W_1 x_i + \sigma_c(l_c^{t-1}) + \sigma_d(l_d^{t-1}))$
10 $\quad$ **end**
11 **end**
12 return $\widetilde{\mu} = W_2(\Sigma_{i \in X} \widetilde{\mu}_i^{(T)})$

b) Learning Parameters. Paired embedding vectors (e.g., $\widetilde{\mu}, \widetilde{\nu}$) are obtained through two identical LSFG-based embedding generation networks with two LSFGs (e.g., g_1 and g_2) as inputs. The output of the whole network represents the similarity of the two functions and is measured by the *Cosine* function denoted as $\widehat{y} = cos(\widetilde{u}, \widetilde{v}) = (\widetilde{u} \cdot \widetilde{v})/(\|\widetilde{u}\| \cdot \|\widetilde{v}\|)$, where $\widehat{y}$ is the predicted similarity output of two functions, ranging from -1 to 1. Given the ground truth $y \in \{1, -1\}$ of LSFGs g_1 and g_2, $y = 1$ indicates they are similar functions; otherwise, they are dissimilar. Suppose that the training

data set has M pairs of labeled samples $\langle g_1, g_2, y \rangle$, then our training objective is to minimize the training errors. We use the stochastic gradient descent algorithm to minimize the error function $min\, E(W_1, W_2, P_1 \cdots Pn, Q_1 \cdots Q_n) = \sum_{m=1}^{M} (\widehat{y} - y)^2$ and obtain the most appropriate model parameters (e.g., W_1, P_1).

3.2 Semantic Emulation

The primary goal of semantic emulation is to sort the M candidate functions obtained from the semantic learning module. By exploiting three steps to obtain more accurate results, semantic emulation reduces the burden of engineers when manually inspecting the M functions: argument recognition, function emulation, and similarity calculation.

3.2.1 Argument Recognition

Before emulating the execution of a function, *BinSeeker* first needs to recognize the required function arguments, which are usually classified as register arguments and stack arguments. With the aid of *IDA Pro* [20], argument recognition is implemented based on the disassembled assembly code from the binary program. For example, the first three arguments of a function may be register arguments stored in the *EAX*, *EDX* and *ECX* registers in the *X86* architecture. The remaining arguments are passed through the program stack, whose space grows from the high address to the low address. Each function has a stack pointer indicating the stack start address. When traversing the assembly instructions, if an instruction accesses a stack address that is larger than the stack start address, the offset of the address relative to the stack start address is recorded as a stack argument.

3.2.2 Function Emulation

BinSeeker first generates a set of random integers for argument assignment. For each function, the same random integer sequence is assigned to identified argument registers and stack offsets in turn. To facilitate the emulation of multiple instruction sets, *BinSeeker* converts each assembly function into the semantics-preserving *VEX-IR* (intermediate representation) [23]. As shown at the top of Fig. 5, for a single machine instruction, the conversion process may generate multiple consecutive *VEX-IR* statements. Unlike a machine instruction that may have multiple consecutive semantic operations, each VEX-IR statement has only one operation and applies to multiple instruction sets. Through emulating the execution of the function on the *VEX-IR* based on the assigned argument values, we extract unified semantic signatures for the binary functions of different instruction sets.

During the emulation execution, *BinSeeker* emulates each function individually and records the dynamic execution trace of the function, which we call its semantic signature. When *BinSeeker* encounters a call to function B while emulating function A, it also enters function B and records the semantic signature of function B in function A. This solves the predictive barrier of function inlining to the semantic learning module. In addition, *BinSeeker* excludes the `main` function to prevent it from encompassing the entire program. When mutual recursion is encountered, we set a threshold beyond which the function will not be entered again to limit the times for executions of the recursion. When the same loop is emulated more than the threshold, we reverse the current branch condition to exit the loop and continue the subsequent emulation. The threshold can be dynamically set during emulating, and we refer to *CACompare* [6] to set the threshold for a fair comparison.

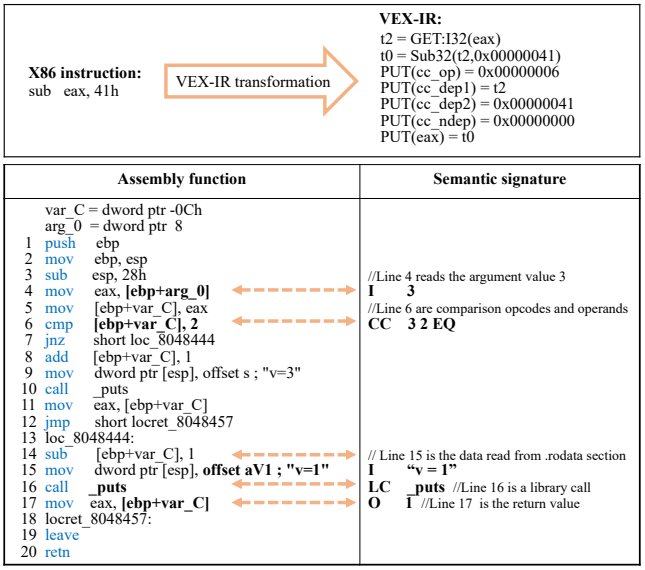


Fig. 5. The example assembly function and its semantic signature.

The semantic signature consists of four parts: *input values*, *output values*, *comparison opcodes/operands*, and *library function calls*. The bottom of Fig. 5 illustrates an assembly function and its semantic signature generated at the end of the emulation. Here the sample function contains only one stack argument named *arg_0*, and its corresponding memory location *'[ebp+arg_0]'* is assigned a value of 3. *Input values* contain the data read from both the assigned argument values and the data sections (e.g., *.rodata*, *.data*). The instructions in Lines 4 and 15 of Fig. 5 contain data reads, and their semantic information is marked as "*I value*". *Output values* consisLt of the return value and memory write values whose addresses are outside the range of the function stack. Line 17 in Fig. 5 is the output value of the function when 3 is used as the function argument. The output values are represented as "*O value*" in the semantic signature. *Comparison opcodes* refer to the condition that controls the jump of basic blocks, and *comparison operands* mean the two values used for comparison. An example is presented in Line 6 of Fig. 5, denoted as "*CC operands opcode*". *Library function calls* record the uses of C language standard library functions during function emulation. Its semantic information is recorded as "*LC name*", such as the Line 16 in Fig. 5.

3.2.3 Similarity Calculation

After obtaining semantic signatures of the vulnerable function and the top-M candidate functions, *BinSeeker* uses the Jaccard similarity coefficient to calculate the similarity score as follows: $J(A, B) = |A \bigcap B| / |A \bigcup B|$, where A and B are semantic signature sequences of the vulnerable function and the target function. By descending the similarity scores, *BinSeeker* reorders the top-M candidate functions and outputs more accurate top-N functions as the final suspected vulnerable functions.

3.3 Implementation

For the semantic learning module, we use *IDAPython* provided by *IDA Pro* [20] to create the CFG for each binary function and extract features for each basic block. Based on the CFG, we infer whether there should be a data dependence edge between two basic blocks by leveraging the *LLVM IR* plugin [24] on *IDA Pro*. We use *TensorFlow* [25] to implement the customized neural network and apply the stochastic gradient descent algorithm to automatically learn model parameters.

For the semantic emulation module, we use the decompiler of the *IDA Pro* for the X86 and ARM binary. For the MIPS binary, we use *IDAPython* to traverse CFG paths to obtain register arguments approximately according to the function calling convention. The key idea is to treat registers that are not written before the first read in the function as register arguments. We also employ *IDAPython* to traverse assembly instructions of the function to determine the address offsets which are recorded as stack arguments. Then we utilize *PyVEX* [26] to convert each assembly function into the *VEX-IR* representation [23]. *BinSeeker* emulates each binary function on the *VEX-IR* representation. After emulating each function contained in the target binary, we record its semantic signature in a separate file. But for the signatures of vulnerable functions, we store them in a custom data structure into a MongoDB database [27] to repeatedly perform search tasks.

4 Experimental Evaluation

BinSeeker is based on the similarity of semantic representation (namely, embedding vector in semantic learning and signature in semantic emulation) to complete cross-platform binary vulnerability search. We compare *BinSeeker* with three most recent and related state-of-the-art cross-platform binary vulnerability search tools: *Genius* [7] and *Gemini* [8] which are semantic learning-based tools, and *CACompare* [6] which is a semantic emulation-based tool. Furthermore, *BinSeeker-*[2] as the front end learning module of *BinSeeker* is also involved in the comparison to demonstrate whether we can replace the front end with Genius or Gemini. Our experiments aim to answer the following two questions:

- RQ1. Is *BinSeeker* more accurate in predicting similar cross-platform binary functions than the other tools?
- RQ2. How efficient is *BinSeeker* in completing a vulnerability search task?

4.1 Experiment Setup

All experiments are performed on an 8-core 3.60GHz Intel i7 machine with 8G memory, an NVIDIA GeForce 1070 GPU and Ubuntu 14.04 LTS operating system.

Data Preparation. In order to mitigate experimenter bias and allow for a fair comparison, we prepare the same two datasets as in the other studies [6], [7], [8] as described below to complete different evaluation tasks. Dataset I is used to train the *BinSeeker-* semantic learning model and to directly compare *BinSeeker-* with *Gemini* to explore whether the proposed *BinSeeker-* model can achieve better similar code prediction accuracy. Dataset II is used to evaluate the accuracy and efficiency of *BinSeeker* for widely studied vulnerabilities. Since the experiment in the other three tools in comparison involves three architectures, we also include binaries of the same three architectures (*X86*, *ARM* and *MIPS*) for unbiased comparison.

- **Dataset I**. Similar to [8], it includes a set of binaries compiled from three open-source software: *BusyBox* (v1.21.0), *OpenSSL* (v1.0.1f and v1.0.1u) and *Coreutils* (v6.5 and v6.7). We use two compilers (*GCC* v4.9 and *Clang* v3.4) with four optimization configurations (*O0–O3*) to compile these programs to three architectures. As a result, we get a total of 368,256 functions and 4,673K basic blocks.
- **Dataset II**. We select widely-used real-world programs from those favored for evaluation by other tools in comparison as shown in Table 2, such as *OpenSSL* and *Coreutils* from *Genius* [7], *curl* and *Wget* from *CACompare* [6]. The vulnerabilities in these programs differ greatly in CFGs across platforms and are difficult to detect accurately [6], [7], [8]. Each program is compiled into four optimization level versions (*O0–O3*) of three architectures using two compilers (*GCC* v4.9 and *Clang* v3.4).

TABLE 2
Dataset II: Open-source programs for vulnerability search

CVE No.	Program	Module	Version
2018-11212	libjepg	jmemmgr	9a
2018-11213 2018-11214	libjepg libjepg	rdppm	
2018-0494 2017-6508	Wget	wget	1.19.1
2015-1791 2014-3508 2016-6302 2016-6303 2016-2842	OpenSSL	openssl	1.0.1f
2014-9471	Coreutils	date	8.13
2017-7407	curl	curl	7.53.1
2015-3237 2015-3145 2015-3144	curl	libcurl	7.40.0

Ground Truth. The training of the customized semantic learning model requires a large number of labeled samples of similar and dissimilar binary function pairs. We use the following strategies to automatically label the samples in the dataset I. With the source code of the function f, we compile it into a set of binary functions denoted as $set(f) = \{f_1, f_2, \cdots, f_n\}$ across different implementation platforms. For each function f_i in $set(f)$, we randomly select a different function $f_j, i \neq j$ to make up a similar sample, and label them as $\langle f_i, f_j, +1\rangle$. We also randomly select another binary function s_k that is not in $set(f)$ to construct a dissimilar sample, and label them as $\langle f_i, s_k, -1\rangle$. A total of 2,761,920 pairs of samples are constructed, where the number of similar sample pairs is half, and no two identical pairs of samples exist.

Training Details. We apply 10-fold cross-validation to train and evaluate *BinSeeker*. Namely, we partition the samples into 10 subsets, each time nine subsets are used to

2. *BinSeeker-* is similar to our previous implementation *VulSeeker* [9]. *VulSeeker-Pro* [12] is not included in the comparison experiments because it focuses on the optimization of a single architecture.

train a model, and one subset is chosen as the test set. We repeat this 10 times and each time the picked test subset is different. The reported result is averaged over 10 times. When training the model, 100,000 pairs of samples in nine subsets are randomly selected for training in each epoch. After finishing each epoch, we randomly shuffle the training set. Note that we refer to the learning component in *BinSeeker* as *BinSeeker-*. That is, *BinSeeker-* produces M candidate functions based on similarity to the function with a known vulnerability.

Default Configuration. Based on our experiments (discussed in Section 4.5), we set up the hyper-parameters of semantic learning module as follows: the training epoch is 100, the learning rate is 0.0001, the embedding depth n is 2, the embedding size p is 64, the number of iterations T for each basic block is 6 and the size of mini-batch is 10. For the semantic emulation module, the values of M and N are set to 200 and 25, respectively.

4.2 Accuracy of Vulnerability Search

Here we mainly answer the RQ1 and focus on whether *BinSeeker* can identify vulnerabilities across platforms more accurately than other tools. For each search of the vulnerable function, we obtain the 200 candidate functions from *BinSeeker-* and finally choose top-25 functions most likely to have the same vulnerability.

4.2.1 Overall Result

We use the vulnerable functions in the *X86-GCC-O3* version of binaries as the source, and the goal of *BinSeeker* is to identify functions from the other versions of binaries that have the same vulnerability. In our experiments, we perform 23 different searches for each vulnerability in dataset II, which results in 345 different searches in total. Columns 2–6 in Table 3 show the search ranking of each vulnerability, and each cell is the average ranking on the 23 different searches[3], such as X86-Clang-O0 and MIPS-Clang-O3.

In Table 3, we first use the top-k metric to measure the search accuracy of the vulnerable function. For the 15 vulnerabilities, we count how many times each tool can rank the real vulnerable function in the top-k candidate list and compute their corresponding percentage for each tool in comparison, where the results are recorded in Rows 17–20 and the '@' character is the separator of the number and percentage. From Table 3, for different values of k, the number of vulnerabilities identified by *BinSeeker* is significantly more than that of the compared tools. Specifically, for *Genius*, *Gemini* and *CACompare*, there are only 3, 1 and 8 vulnerabilities ranking in the top-3 candidate list, which results in 20.00%, 6.67% and 53.33% top-3 accuracy, respectively. On the other hand, *BinSeeker* identifies 11 real vulnerabilities and achieves a 73.33% top-3 accuracy rate. Similarly, the top-5 accuracy for *Genius*, *Gemini* and *CACompare* are merely 33.33%, 13.33% and 53.33%, respectively, whereas for *BinSeeker* that is 93.33%. All vulnerabilities are ranked in the top-20 candidate list by *BinSeeker*. In contrast, the values for the other three tools are 8, 5 and 10, respectively.

In statistics, MRR (mean reciprocal rank) is commonly used to measure the evaluation of the target search results ordered by the probability of correctness, where reciprocal rank value is the multiplicative inverse of the ranking of the first correct result. In our search scenario, there is only one correct result per search. So we also use this indicator ($MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}, |Q| = 15$ here) to measure the effectiveness of the four tools. When each search result is ranked first, the MRR reaches a maximum value of 1. From Table 3, we can see that *BinSeeker* has an MRR of 0.65, which is much larger than that of the other four tools.

3. We have split Table 3 and listed search rankings according to the architectures, compilers, and optimization options in detail on the website https://github.com/PaperData/TSE_data.

TABLE 3
The accuracy of five tools for comparison on 15 vulnerabilities.

CVE No.	*Genius*	*Gemini*	*CACompare*	*BinSeeker-*	*BinSeeker*
2018-11212	5	79	1	5	1
2018-11213	3	4	1	3	1
2018-11214	3	17	1	3	1
2018-0494	9	75	3	72	1
2017-6508	10	17	68	14	2
2015-1791	48	236	67	189	4
2014-3508	96	58	60	45	2
2016-6302	50	418	19	149	5
2016-6303	78	328	236	175	12
2016-2842	149	392	1	197	1
2014-9471	1	2	2	2	2
2017-7407	5	7	3	3	2
2015-3237	23	28	1	30	1
2015-3145	8	149	40	38	5
2015-3144	69	503	16	105	1
Top-1 (#@%)	**1@6.67**	**0@0**	**5@33.33**	**0@0**	**7@46.67**
Top-3 (#@%)	**3@20.00**	**1@6.67**	**8@53.33**	**4@26.67**	**11@73.33**
Top-5 (#@%)	**5@33.33**	**2@13.33**	**8@53.33**	**5@33.33**	**14@93.33**
Top-20 (#@%)	**8@53.33**	**5@33.33**	**10@66.67**	**6@40**	**100@100**
MRR	**0.17**	**0.07**	**0.42**	**0.13**	**0.65**

4.2.2 Result Analysis

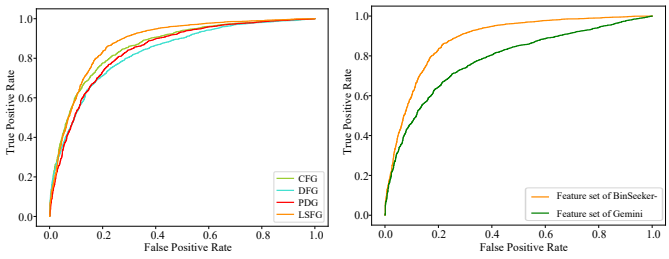
The ranking in Table 3 shows that the two learning-based approaches *Genius* and *Gemini* rank four and nine vulnerabilities beyond 50, respectively. We have examined the assembly functions with the corresponding vulnerability and found the reasons that can cause the inaccuracy. One is that function inlining exists in the higher optimization level binary functions during compiling, which affects instruction features of the function. The other reason is that the CFGs of the same function are changeable under different platforms, which is reflected in the semantic embedding vector of the function. These lead to inaccurate prediction results of these two tools. Semantic emulation-based approach *CACompare* also ranks 4 vulnerabilities beyond 50. Because the illegal arguments passed to the functions cause the body of the function to be bypassed during emulation, unrepresentative semantic signatures are produced. In addition, constant integers in programs are treated as memory references to constant data sections (e.g., *.rodata* section) in one platform, and is directly used as immediate values in another platform. For binary functions from the same source function, the instruction addressing patterns and the order of instruction accessing memory differ greatly. This also affects the order and number of semantic signatures. These cause the phenomenon of mismatching.

BinSeeker improves the search accuracy of *Gemini* by introducing LSFG in the semantic learning component. Then *BinSeeker* executes semantic emulation on similar functions to further improve the search accuracy of the vulnerability. This is why *BinSeeker* can achieve the best results. How-

ever, one function with the targeted vulnerability is ranking outside the top-10 candidate results in Table 3. The reasons that reduce the accuracy of *CACompare* also have effects on *BinSeeker*, since they all involve the process of function emulation. Fortunately, *BinSeeker* only emulates M functions, thus to a certain extent avoiding the shortcomings that affect search accuracy of *CACompare*. Due to the combination of semantic learning and emulation, *BinSeeker* can make use of the advantages of both and compensate for the corresponding disadvantages. As a result, *BinSeeker* achieves an ideal search ranking in most cases, and a somewhat poor ranking in rare cases.

4.2.3 Contributions of LSFG and Feature Set

We propose a new semantic learning approach in *BinSeeker*, which has a 1.23× ranking improvement when compared to the learning-based approach *Gemini*. It has a similar process to *Gemini*, but differs in two aspects. The first difference is that we also add the DFG between basic blocks instead of just using the CFG. The other difference is that we use a set of different features to participate in generating function-level embedding vectors. Thus we conducted experiments on dataset I to verify the contributions of these two improvements to search accuracy improvement. Based on the default configuration detailed in Section 4.1, we train our models on the training set according to the following two different requirements, then evaluate the effectiveness on the test set.



(a) Effectiveness of LSFG (b) Effectiveness of feature set

Fig. 6. Contributions of LSFG and feature set in *BinSeeker*

a) Effectiveness of LSFG. After fixing the feature set of *BinSeeker-*, we use CFG, DFG, PDG and LSFG separately for the experiments. Fig. 6(a) shows their effectiveness in the form of ROC curve (receiver operating characteristic curve). The area enclosed by the ROC curve and the x-axis is expressed as AUC (area under the curve), which is equivalent to the probability that a randomly chosen positive example is ranked higher than a randomly chosen negative example. The AUC values for CFG, DFG, PDG and LSFG are 0.86, 0.83, 0.85 and 0.88, respectively. When the model achieves high classification accuracy, even small improvements are not trivial. We conclude that applying CFG alone in the semantic learning module can achieve better search accuracy than that of DFG and PDG. It means that the control flow structure of the function is better at identifying similar functions than data transfer in our semantic learning module. However, by adding DFG, the search accuracy will be further improved. This proves that the LSFG used in *BinSeeker* is effective and achieves a 2.33% improvement than CFG alone.

b) Effectiveness of Feature Set. Related researches have proposed several sets of block-level features for predicting similar functions in learning-based methods [7], [22]. By observing and analyzing the characteristics of different binaries compiled across different platforms, we propose a set of features that are suitable for performing the vulnerability search task. We use the feature set of *Gemini* and feature set of *BinSeeker-* to conduct experiments for code similarity prediction, both of which are based on LSFG. Fig. 6(b) shows the performance of different feature sets. We observe that the feature set of *BinSeeker-* has better performance compared to the feature set of *Gemini*. The AUC value for *BinSeeker-*'s feature set is 0.88, which is 12.8% higher than the feature set of *Gemini*. The experiment shows that the proposed feature set in Section 3.1.2 is more robust and change little under various implementation platforms with different microprocessor architectures and various compilation optimization configurations.

4.3 Time Cost of Vulnerability Search

We discuss search time obtained by using dataset II and training time which is based on the dataset I, respectively.

Search Time. We answer RQ2 about how much time *BinSeeker* needs to complete a vulnerability search task. This will have a direct bearing on whether *BinSeeker* can be used in the industry. We use the *X86-GCC-O0* version of the program as the target to experiment for time cost. Table 4 shows the time cost of the five tools for each program. Column 2 lists the number of functions in each program. Columns 3–7 are the time cost to complete a vulnerability search. All the time is measured in seconds, and the values are rounded.

TABLE 4
Time cost of the five tools on each program. (Unit: **second**)

Program	#Functions	*Genius*	*Gemini*	*CACompare*	*BinSeeker-*	*BinSeeker*
libjpeg	580	928	81	343	110	206
Wget	804	1,326	116	531	160	282
Openssl	5,995	8,992	849	8,432	1,145	1,323
Coreutils	119	198	20	92	23	44
curl-7.53.1	1,113	1,747	167	722	225	362
curl-7.40.0	2,760	4,664	469	1,022	549	805
AVG	**1,895**	**2,976**	**284**	**1,857**	**369**	**504**

In this experiment, the program contains an average of 1,895 functions. *Gemini*, *BinSeeker-* and *BinSeeker* demand less time cost in completing a vulnerability search, which are 284s, 369s and 504s on average, respectively. It means that they need an average of 0.15s, 0.19s and 0.27s to calculate the similarity between a target function and a vulnerable function. However, *Genius* and *CACompare* require 2,976s and 1,857s to finish one vulnerability search from 1,895 functions. Their time costs are much higher and require 4.90× and 2.68× more than that of *BinSeeker*. As a result, *BinSeeker* is clearly better suited to perform the vulnerability search task on a large scale of code.

Looking closely at Columns 2 through 7 of Table 4, we find that the time cost of *Genius*, *Gemini* and *BinSeeker-* increases linearly with the number of functions roughly. The main reason is that these three tools need to extract features, generate semantic embedding vectors, and compute the similarity to the vulnerability for all functions of the program. However, the time cost of *CACompare* and *BinSeeker* does not follow the same pattern. *CACompare* emulates each

function dynamically, so the number of loop executions in the function affects the time cost, which does not conform to the linear growth phenomenon. In contrast to that, *BinSeeker* only needs to perform the emulation for the fixed number of candidates. Therefore, the more functions are in the program, the closer the time spent by *BinSeeker* is to *BinSeeker-*.

In summary, for a single function, although the time cost of *BinSeeker* is 0.12s more than the fastest tool *Gemini* on average, *BinSeeker* manages to achieve a better search accuracy in a reasonable amount of time. Although *Genius* is also a learning-based approach, it is nearly 10× slower than *Gemini* or *BinSeeker-* and thus inappropriate to be the semantic learning module of *BinSeeker*.

Training time. Since only the learning-based approaches require training models, we describe the training time for three tools in comparison: *Genius*, *Gemini*, and *BinSeeker-*. With 100,000 pairs of samples, *Genius* requires 50 days to produce a model by spectral clustering. To train models for 100 epochs under our default configurations, *Gemini* and *BinSeeker-* require 17 and 22 days, respectively. Nevertheless, the training process is a one-time cost without any effect on vulnerability search efficiency after deployment. It means that once obtaining an effective model, we can use it in any appropriate scenario without having to retrain it again. As we will see in Section 4.5.4, we do not need to train for 100 epochs. The 50-epoch model can be as good as the 100-epoch model, and the AUC value increases very slowly as the epoch rises so that we can reduce the training time of *BinSeeker-* by half (specifically, 11 days).

4.4 Effectiveness of *BinSeeker-* Front End

The search accuracy experiment in Section 4.2 shows that *Genius* performs slightly better than *BinSeeker-*. The first intuition is whether we can use *Genius* as the front end of *BinSeeker* to get candidate lists for known vulnerabilities. Therefore, in this section, we will evaluate this thought from the aspects of accuracy and efficiency by analyzing the experimental data. The role of the front end is to output a list of candidate functions for further refinement ordering by the back end. Therefore, changing the value of M will affect whether the output candidate list of the front end can actually contain known vulnerabilities. The experiments mainly focus on combining *Genius* and *BinSeeker-* with the semantic emulation back end respectively to evaluate the recall rate and time cost, abbreviated as *Genius + Emulation* vs. *BinSeeker*. Since the only difference is the front end, we only change the value of M, and the other configurations are the same as those in Sections 4.2 and 4.3.

The effectiveness of the front end in *BinSeeker* can be evaluated via the recall rate, which measures the ability of the search system to find real vulnerabilities. In this experiment, the value of N is fixed to 20, M has four values (that is 50, 100, 150, and 200), and the front-end output is denoted as top-M candidates. Let TP be the number of true vulnerabilities ranking in the top-M candidates (true-positives) and FN be the number of true vulnerabilities ranking outside the top-M candidates (false-negatives). For the false-negative case, *BinSeeker* would not identify them correctly. The metric $recall\ rate = \frac{TP}{(TP+FN)}$ reflects the completeness of the searched positives on the top-M candidates. Table 5 shows the recall rates with different M values and the average time cost used to calculate each pair of vulnerable function and target function.

TABLE 5
The recall rate on different M values and the average time cost.

Tools	Recall Rate, N=20				Time Cost
	M=50	M=100	M=150	M=200	
Genius+Emulation	73.33	93.33	100.00	100.00	1.57
BinSeeker	60.00	66.67	80.00	100.00	0.19

As can be seen from Table 5, when N is fixed at 20, different M values produce diverse recall rates for the two kinds of combinations between two front ends and one back end. When M is 50, the combination of *Genius* plus emulation can generate a recall rate of 73.33%, which can identify 22.22% more real vulnerabilities than *BinSeeker*. When M expands to 100, the recall rate of the former reaches 93.33% and improves 39.99% than *BinSeeker*. However, when M increases to 200, the recall rate of both reaches 100% in dataset II. Once the target binary is determined, the total number of binary functions to be processed by each front end is the same and does not change with M. So the average time cost of processing each pair of functions is also an important factor in choosing a suitable front end. *BinSeeker*'s front end only needs 0.17s to handle a pair of functions, while *Genius* takes 7.26 times more time. This means assuming that a binary has 5,000 functions, it takes *Genius* and *BinSeeker-* 2.18 hours and 0.26 hours to output top-M candidate functions to the emulation module, respectively. In addition, model training time for *Genius* is 2.27 times that of *BinSeeker-*.

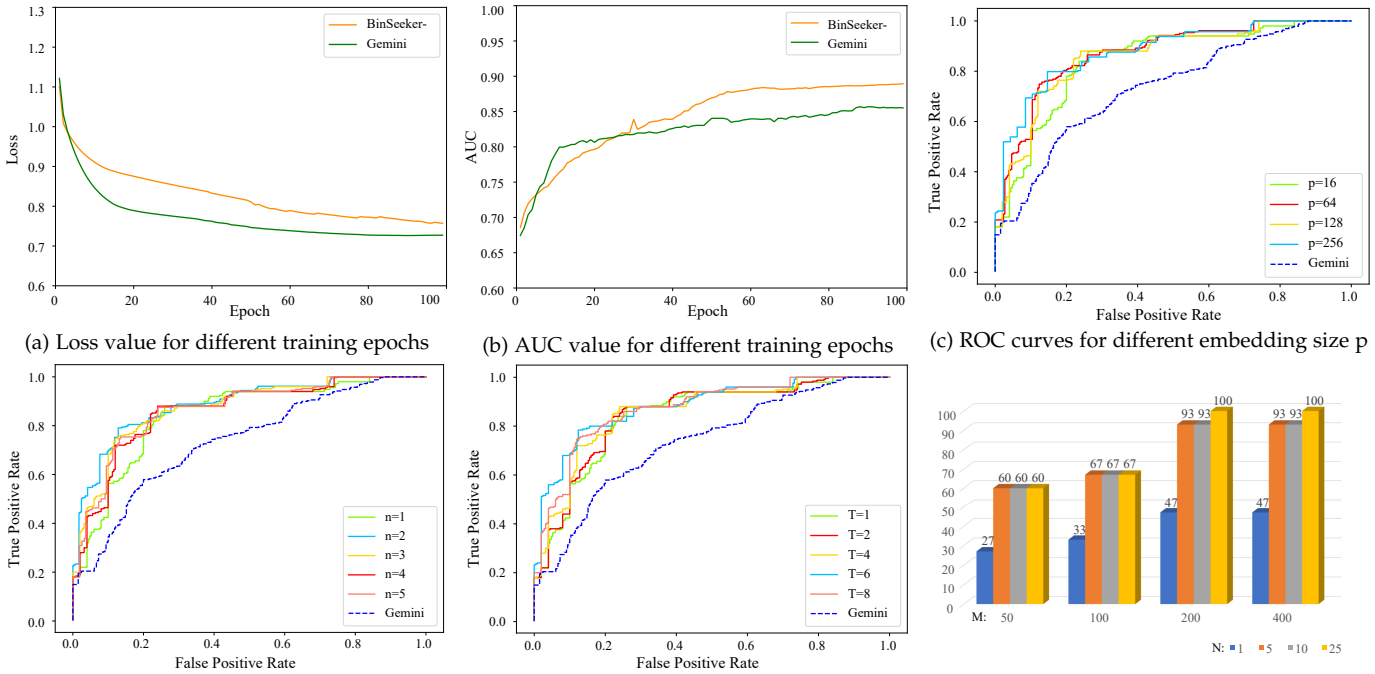
As a summary, when the total number of functions to be searched is determined, the front end that we need to choose should take as little time as possible to gain the maximum recall rate. Obviously, *BinSeeker-* front end with slightly larger M value is more suitable than Genius here.

4.5 Hyper-parameters Studies

In this section, we explore how to set the most appropriate parameters for *BinSeeker* and evaluate the impacts of different hyper-parameters on search accuracy. Studies contain two parts: the parameter settings of semantic learning module *BinSeeker-* with dataset I, the selection of M and N values in *BinSeeker* with dataset II. The first part is from Section 4.5.1 to Section 4.5.4, where all the hyper-parameters are evaluated on the test set. In the study of each hyper-parameter, except for the parameter being evaluated, other parameters take default settings, as described in Section 4.1. The second part is discussed in Section 4.5.5.

4.5.1 Size of Training Epochs

Increasing training epochs results in more time being spent in weight updating. However, it is pointless to increase the number of training epochs blindly, which may not substantially change the parameter values of the model. So we want to know when the performance of the model tends to be stable. In total, we train the model for 100 epochs and evaluate the loss value and AUC value on the test set for every epoch. The loss value refers to the sum of the squares of the difference between the predicted similarity and the label value among all samples. Fig. 7(a) and Fig. 7(b) show the loss value and the AUC value, respectively. We can see that our approach has achieved a good performance in about 50 epochs, the AUC value is 0.88, and the loss value is



(a) Loss value for different training epochs

(b) AUC value for different training epochs

(c) ROC curves for different embedding size p

(d) ROC curves for the embedding depth n

(e) ROC curves for the number of iterations T

(f) Percentage for different M and N values

Fig. 7. Hyper-parameter studies results. Employing the test set of dataset I, Fig. 7(a–e) describe the predictive effects of the semantic learning model with different choices of the training epochs, the embedding size, the embedding depth and the number of iterations. Using dataset II, Fig. 7(f) shows the percentage of vulnerabilities that *BinSeeker* can successfully find under different M and N values. With the variation of embedding size, the embedding depth and the number of iterations, the AUC value of *BinSeeker* remains higher than that of Gemini. The temporary exception in Fig. 6(b) is because our model contains more parameters and needs more epochs for training in the early stage.

0.83. The loss values for both approaches are below 1.2. As the number of epochs increases, we eventually have a higher AUC value than Gemini, even though it is lower in about the first 30 epochs. The main reason is that our model contains more parameters and needs more epochs for training in the early stage.

4.5.2 Embedding Size

The embedding size refers to the dimension of the embedding vector used to represent the function semantics. We use the ROC curves to evaluate which embedding size can achieve the best performance. Fig. 7(c) plots the experimental results. When the embedding size is greater than 64, their corresponding ROC curves are close to each other. We choose 64 as the default embedding size because it can reduce the time cost of training and prediction. Whatever embedding size is set up for *BinSeeker*, the AUC value is higher than that of *Gemini* with the optimal settings. This phenomenon also applies to the embedding depth and the number of iterations.

4.5.3 Embedding Depth

The embedding depth refers to the number of layers of the two fully-connected networks represented as σ_c, σ_d. Fig. 7(d) shows the effects of varying embedding depth. The relatively good AUC value is obtained when the embedding depth is 2. This means that by increasing a two-layer fully-connected network, the generated embedding vectors also get higher representation capabilities and better capture the function semantics. However, when the embedding depth exceeds 2, there will be no benefit other than a higher time cost of training and prediction.

4.5.4 Number of Iterations

It refers to the number of hidden layers in the LSFG-based embedding generation network in Fig. 4(b). We vary the number of iterations T and get results of ROC curves drawn in Fig. 7(e). When the number of iterations is 6, our approach achieves the best performance of code clone. This means that the feature vector of each vertex in LSFG can propagate 6-hops along with the graph topology.

4.5.5 Selection of M and N Values

The performance of *BinSeeker* depends on the M candidate functions produced by the semantic learning module. This group of experiments will discuss the influence of M values on the value of N in *BinSeeker*. Fig. 7(f) shows the effectiveness of different M and N values. The x-axis is the different M values (50, 100, 200, and 400). The y-axis is the percentage of vulnerabilities that *BinSeeker* can successfully find when the M and N values are fixed. For each M, we study the percentage of four N values (1, 5, 10, and 25).

From Fig. 7(f), we know that when N is fixed, the number of vulnerabilities *BinSeeker* can detect increases with the increase of the M value. But the N value needs to be at least 25 to achieve 100% search accuracy. Larger M value is more likely to ensure that the vulnerability can be searched, which is cost-effective to increase the time cost of just 0.98s per function on average. When the M value is fixed, the percentage also gets higher with the increase of the N value. When M is 200, the percentage can reach 100%. However, it may not grow to 100% due to a small M value that will result in no vulnerability in the top-M candidate results. To sum up, the values of M and N are best set to 200 and 25 for all vulnerabilities to be found within a relatively short time.

5 Threats to Validity

We present some potential threats that may affect the performance of *BinSeeker* and provide some coping strategies.

a) Semantic Learning Model. The performance of *BinSeeker* depends on the top-M candidate functions exported by the semantic learning module. If we can reduce the value of M without losing accuracy, the total vulnerability search time required by *BinSeeker* will be shortened. When training the network model, we can enhance the generalization ability of the model by increasing the discrete training samples from multiple binary programs. In general, the model with a larger training epoch will have stronger vulnerability prediction ability. Nevertheless, we need to pay attention to model over-fitting.

b) Function Inlining. Compiler optimizations may inline some functions to the callers to maximize the runtime speed. Whether the analyzed function contains inlined functions affects the number of instructions within the function, and ultimately affects function embedding vectors. Bingo [4] proposes an inline decision algorithm based on six commonly-observed invocation patterns. It is possible to adopt the algorithm to inline called functions selectively, then to extract basic block features. In this way, analyzed cross-platform functions will have a consistent inline strategy, which is likely to increase search accuracy.

c) Binary Obfuscation. Code obfuscation can significantly change the control flow structure (e.g., CFG flattening, opaque predicates), sometimes even the data dependence structure, which brings enormous side effects to the function semantics. Before constructing the LSFG for learning and emulating, we first need to deobfuscate code [28], which will effectively solve the impact of obfuscation.

d) Binary Diversity. The binary functions obtained from the same source code under different compilation scenarios have great differences. For some less complex compilation scenarios, the accuracy of *Gemini* and *BinSeeker* can be much higher than the results presented in Fig. 7(b). For example, in the dataset settings of [8], *Gemini*'s AUC value of the model is 0.971 and ours is about 0.984, which is much higher than the value of 0.818 and 0.885 tested in our test set. Our dataset I contains five different programs of various sizes with more compilation options, but the dataset of *Gemini* only contains two. We find that the more complex the compilation scenario is, the greater the improvement *BinSeeker* achieves. The emulation approach described in this paper mitigates the impact of binary diversity to some extent.

e) False Negative. If the front end *BinSeeker-* gets some false negative, the second part is useless. But both learning-based and emulation-based approaches have the problem of false negatives and false positives. The false positives of learning-based approaches are more serious, and the false negatives of emulation-based approaches are more serious. To our best knowledge, it is not possible to completely solve the problem of false positives and false negatives currently, and there is no exhaustive solution for every single one. We can increase the parameter M to reduce the possibility of false negative cases, as described in Section 4.5.5. In the case of the original Gemini, if we consider a function with vulnerability ranked top-5 as accurate, Gemini only achieves an accuracy rate of 13.33%. When the values of M and N are 100 and 5, the false negatives of BinSeeker is only 33%, and the top-5 accuracy is 67%. From the figure, we know that when N is fixed, the number of vulnerabilities BinSeeker can detect increases with the increase of the M value. Larger M value is more likely to ensure that the vulnerability can be in the candidate list, which is cost-effective to increase the time cost of just 0.98s per function on average. In our experiments, when the values of M and N are set to 200 and 25, all vulnerabilities can be found in the top-25 results. This is a significant breakthrough in reducing the human efforts of manually confirming real vulnerabilities from a large number of suspicious functions.

6 Related Work

6.1 Syntax and Structure Based Search

The idea of vulnerability search based on syntax and structure is that similar code fragments have similar syntax structures. *CCFinder* [29] detects cloned source code in large scales based on lexical code tokens. It generates the token sequences of the source code through a lexical analyzer and then obtains the regularized sequences with rule-based transformation. Finally, it applies a suffix-tree matching algorithm to compute similarity. *BinDiff* [30] builds CFGs of the two binaries and then adopts a heuristic algorithm to normalize and match the two CFGs. *BinSlayer* [31] improves *BinDiff* by adopting the *Hungarian algorithm* [32] for bipartite graph matching. *DECKARD* [33] produces an abstract syntax tree (AST) to represent the source program, and further extracts feature vectors from AST, improving the efficiency and accuracy of the detection.

6.2 Semantics Calculation Based Search

Semantics calculation based vulnerability search approaches use the semantic features calculated from the syntactic structures to better represent searched codes. *COP* [10] is a plagiarism detection tool that combines program semantics with the longest common sub-sequence based fuzzy matching. *BinHunt* [34] considers matching CFGs as the maximum common induced sub-graph isomorphic problem. It leverages symbolic execution and theorem proving to match the basic blocks with the same semantics. *BLEX* [35] is a dynamic function matching tool that uses several semantic features obtained during the function execution (for example, values read from the program heap) in the matching process. *BinGold* [22] extracts the semantics of binary code concerning both data and control flow and synthesizes them into a novel representation called the semantic flow graph. However, it does not support cross-architecture clone detection, and its average precision is 74.97%. *BinSim* [5] calculates the equivalences of aligned system calls to better handle code obfuscation. It is a hybrid method to identify fine-grained semantic similarities or differences between two execution traces.

6.3 Learning Based Search

Learning-based vulnerability search approaches automatically learn semantic features of the program or select proper code similarity algorithms to find the cloned code. *VulPecker* [36] applies the SVM classification approach to select a set of code-similarity algorithms that could distinguish unpatched pieces of code from the patched ones. *VulDeePecker* [37] uses code gadgets to represent programs and employs BLSTM

neural network to extract features instead of having human experts manually defining the features. In [38], the authors present a deep learning-based clone detection tool, extracting hidden patterns of the lexical and syntactic levels based on the RNN. Based on the bipartite graph matching algorithm, *Genius* [7] calculates the similarity between a specified ACFG (attributed control flow graph) and each representative ACFG in the codebook generated by the spectral clustering algorithm. *Gemini* [8] generates an embedding vector for each function represented by the CFG, then compares each pair of vectors to get the prediction result. *VulSeeker* [9] proposes a set of lightweight instruction features and integrates DFG into CFG to enhance the robustness against structural differences in the CFG. *VulSeeker-Pro* [12] supplements multiple semantic signatures to evaluate function similarity, and mainly focuses on the optimization of a single architecture. *BinSeeker* seamlessly integrates our previous optimization [9], [12], and supports the cross-architecture function emulation to achieve a better performance in both time and accuracy.

7 Conclusion

In this paper, we present *BinSeeker*, an accurate and efficient cross-platform binary vulnerability seeker that integrates semantic emulation with semantic learning. In semantic learning, by combining both the data flow dependency and the control flow dependency of the binary function, we capture more function semantics than the existing approaches and output the M candidate functions that are similar to the vulnerable function. Then through semantic emulation, *BinSeeker* further improves the search accuracy and outputs more accurate top-N candidate functions out of the M candidates. Overall, *BinSeeker* achieves higher search accuracy with a lower computation requirement than state-of-the-art tools such as *Genius*, *Gemini* and *CACompare*. Compared to the time users spend in manually identifying real vulnerabilities from a collection of hundreds of false positives and a few true positives, the running time is almost negligible. Our future work will seek to improve the robustness of the semantic emulator and apply it to more platforms.

References

[1] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Software Eng.*, vol. 32, no. 3, pp. 176–192, 2006.

[2] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire OS distributions," in *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, 2012, pp. 48–62.

[3] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[4] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: cross-architecture cross-os binary search," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 678–689.

[5] J. Ming, D. Xu, Y. Jiang, and D. Wu, "Binsim: Trace-based semantic binary diffing via system call sliced segment equivalence checking," in *USENIX Security Symposium*, 2017.

[6] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, 2017, pp. 88–98.

[7] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, 2016, pp. 480–491.

[8] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 363–376.

[9] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *The 33rd IEEE/ACM International Conference on Automated Software Engineering*, 2018.

[10] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *FSE*, 2014.

[11] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 709–724.

[12] J. Gao, X. Yang, Y. Fu, Y. Jiang, H. Shi, and J.-G. Sun, "Vulseeker-pro: enhanced semantic learning based binary vulnerability seeker with emulation," in *ESEC/SIGSOFT FSE*, 2018.

[13] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a siamese time delay neural network," in *Advances in Neural Information Processing Systems 6, [7th NIPS Conference, Denver, Colorado, USA, 1993]*, 1993, pp. 737–744.

[14] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, 2016, pp. 2702–2711.

[15] CVE, "Common vulnerabilities and exposures," http://cve.mitre.org/, accessed April 4, 2018.

[16] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, PPREW@POPL 2013, Italy*, 2013, pp. 5:1–5:6.

[17] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, pp. 349–360.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," 1984.

[19] D. Caselden, A. Bazhanyuk, M. Payer, S. McCamant, and D. Song, "Hi-cfg: Construction by binary analysis and application to attack polymorphism," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 164–181.

[20] I. Pro, "The ida pro disassembler and debugger," https://www.hex-rays.com/, accessed April 7, 2018.

[21] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 138–157.

[22] S. Alrabaee, L. Wang, and M. Debbabi, "Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs)," *Digital Investigation*, vol. 18, pp. S11–S22, 2016.

[23] J. Seward and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan Conference on Programming Language Design and Implementation*, 2007, pp. 89–100.

[24] MIASM, "Reverse engineering framework," https://github.com/cea-sec/miasm, accessed May 20, 2018.

[25] M. Abadi, A. Agarwal, P. Barham *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from http://tensorflow.org.

[26] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[27] I. MongoDB, "Mongodb database as a service," https://www.mongodb.com, accessed August 7, 2018.

[28] B. Yadegari, "Automatic deobfuscation and reverse engineering of obfuscated code," Ph.D. dissertation, University of Arizona, Tucson, USA, 2016.

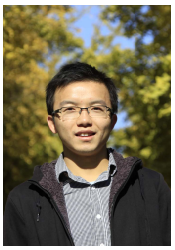
[29] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, pp. 654–670, 2002.

[30] H. Flake, "Structural comparison of executable objects," in *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG*

SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings, 2004, pp. 161–173.
[31] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop 2013, PPREW@POPL 2013, January 26, 2013, Rome, Italy*, pp. 4:1–4:10.
[32] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society for Industrial Applied Mathematics*, vol. 5, no. 1, pp. 32–38, 1957.
[33] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," *29th International Conference on Software Engineering (ICSE'07)*.
[34] D. Gao, M. K. Reiter, and D. X. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, October 20-22, 2008*, 2008, pp. 238–255.
[35] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security Symposium*, 2014.
[36] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, USA*, 2016, pp. 201–213.
[37] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018.
[38] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

Jian Gao received the BS degree in software engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2016. He is currently working toward the Ph.D. degree in software engineering at Tsinghua University, Beijing, China.

His research interests include binary vulnerability search, binary clone detection, machine learning in program analysis and their applications to industry.



Yu Jiang received the BS degree in software engineering from Beijing University of Posts and Telecommunications in 2010, and the PhD degree in computer science from Tsinghua University in 2015. He worked as a Postdoc researcher in the department of computer science of University of Illinois at Urbana-Champaign, IL, USA, in 2016, and is now an assistant professor in Tsinghua University. His current research interests include domain specific modeling, formal computation model, formal verification and their applications in embedded systems.

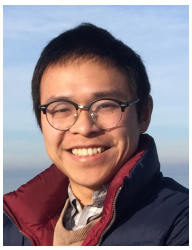
Zhe Liu received his Ph.D degree from the Laboratory of Algorithmics, Cryptology and Security (LACS), University of Luxembourg. His Ph.D thesis has received the prestigious FNR Awards. He is a full professor in College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA) and SnT, University of Luxembourg. He has been a visiting scholar in City University of Hong Kong, COSIC, K. U. Leuven as well as Microsoft Research, Redmond. His research interests include computer arithmetic and information security.

Xin Yang received the BS degree in software engineering from Beijing Jiaotong University, Beijing, China, in 2016. She is currently working toward the M.S. degree in software engineering at Tsinghua University, Beijing, China.

Her research interests include binary vulnerability search, machine learning in program analysis and their applications to industry.

Cong Wang is a PhD student in School of Software, Tsinghua University. He received his B.S. degree in the School of Software, Tsinghua University in 2015. His research interest is program analysis, software testing and deep learning.

His research interests include binary vulnerability search, binary clone detection, machine learning in program analysis and their applications to industry.



Xun Jiao is an assistant professor in the ECE department of Villanova University. He obtained the Ph.D. degree from the department of Computer Science and Engineering at the University of California, San Diego. He received the dual bachelor's degree from the Beijing University of Posts and Telecommunications, China and the Queen Mary University of London, United Kingdom, in 2013. His research interests include error-tolerant computing and machine learning.

Zijiang Yang is a professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.

Jiaguang Sun received the BS degree in automation science from Tsinghua University in 1970. He is currently a professor in Tsinghua University. He is dedicated in teaching and R&D activities in computer graphics, computer-aided design, formal verification of software, and system architecture. He is currently the director of the School of Information Science & Technology and the School of Software in Tsinghua University.