VulSeeker-Pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation

Jian Gao*[†]
School of Software, Tsinghua University
Beijing, China
gaojian094@gmail.com

Xin Yang
School of Software, Tsinghua University
Beijing, China
yangx16@mails.tsinghua.edu.cn

Ying Fu
School of Software, Tsinghua University
Beijing, China
fy17@mails.tsinghua.edu.cn

Yu Jiang[‡]
School of Software, Tsinghua University
Beijing, China
jiangyu198964@126.com

Heyuan Shi
School of Software, Tsinghua University
Beijing, China

Jiaguang Sun
School of Software, Tsinghua University
Beijing, China

ABSTRACT

Learning-based clone detection is widely exploited for binary vulnerability search. Although they solve the problem of high time overhead of traditional dynamic and static search approaches to some extent, their accuracy is limited, and need to manually identify the true positive cases among the top-M search results during the industrial practice.

This paper presents *VulSeeker-Pro*, an enhanced binary vulnerability seeker that integrates function semantic emulation at the back end of semantic learning, to release the engineers from the manual identification work. It first uses the semantic learning based predictor to quickly predict the top-M candidate functions which are the most similar to the vulnerability from the target binary. Then the top-M candidates are fed to the emulation engine to resort, and more accurate top-N candidate functions are obtained. With fast filtering of semantic learning and dynamic trace generation of function semantic emulation, *VulSeeker-Pro* can achieve higher search accuracy with little time overhead. The experimental results on 15 known CVE vulnerabilities involving 6 industry widely used programs show that *VulSeeker-Pro* significantly outperforms the state-of-the-art approaches in terms of accuracy. In a total of 45 searches, *VulSeeker-Pro* finds 40 and 43 real vulnerabilities in the top-1 and top-5 candidate functions, which are 12.33× and 2.58× more than the most recent and related work *Gemini*. In terms of efficiency, it takes 0.22 seconds on average to determine whether the target binary function contains a known vulnerability or not.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**;

KEYWORDS

semantic learning, function emulation, vulnerability search

ACM Reference Format:
Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2018. VulSeeker-Pro: Enhanced Semantic Learning Based Binary Vulnerability Seeker with Emulation. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275524

1 INTRODUCTION

Copy-paste of code and reuse of third-party libraries are common ways to improve the software development efficiency in industry. The risk of such a practice is that unpatched vulnerable code can easily be propagated inadvertently to different programs. Moreover, sometimes source code programs are not easily obtained, leading to vulnerability scanning of binary programs is essential and important. So code clone detection becomes an effective way to discover known vulnerabilities in industrial practice.

Though clone-based techniques [2, 5, 10, 16] and fuzzy testing [9, 15] are effective vulnerability search approaches, their performance is limited for cross-platform binaries. Traditional clone-based vulnerability search approaches can be divided into two categories. Static approaches usually rely on the graph matching algorithm on control flow graphs (CFGs) of functions to identify binary code similar to the vulnerable code [4, 10]. However, they face the challenge that CFGs of the same function differ significantly in different compilation configurations (e.g., *O0-O3* optimization levels), affecting the accuracy of vulnerability prediction across compilation configurations. On the contrary, dynamic approaches overcome the obstacles caused by compilation configurations through monitoring the runtime traces of binary programs in a real operating environment and performing equivalence checking between two traces [7, 11]. But the limitation of such approaches is that it is so time-consuming that it is impractical to perform the vulnerability search in large-scale code in industry.

CFG在大规模预测中不切实际

In recent years, due to low time cost and less domain knowledge requirements, learning-based approaches have been continuously proposed to perform the vulnerability search task for binary programs of industrial circles [5, 16]. They usually use learning-based algorithms to transform the low-level instruction features into high-level semantic features of the function, so these learning-based algorithms can be called **semantic learning**. For example, *Genius*

用机器学习方法检测漏洞

*Also with Beijing National Research Center for Information Science and Technology.
[†]Also with Key Laboratory for Information System Security, Ministry of Education.
[‡]Correspondence author

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275524

[5] utilizes the spectral clustering to train a codebook and obtain the semantic features of a specific function by measuring the similarity between the specific function and the codebook. However, its search accuracy for two case studies on a large set of firmware images is 28% and 48% in the top-50 candidate functions [5], which is not high enough for industry application.

By taking into account the transfer of basic block features along the CFG topology of the function, *Gemini* [16] applies the deep learning approach to further improve the accuracy and efficiency of *Genius* [5]. As a result, it acquires more robust function semantics and achieves a higher search accuracy than *Genius*. However, its accuracy improvement is still not enough. According to our experiments, *Gemini* ranks the vulnerable function 50th on average among the 45 vulnerability searches. So many suspected vulnerabilities predicted by *Gemini* pose a tremendous challenge to program analysts. They have to spend a lot of time identifying real vulnerabilities from a collection of hundreds of false positives and a few true positives. To put forward a binary vulnerability search tool available and user-friendly to the industry, we have to balance the following two major factors:

- **High search accuracy.** The proposed tool should be immune to the structural differences introduced by compilation configurations. This means that vulnerable code should be ranked at the front position of candidate results reported by vulnerability seekers. It will allow analysts to spend less time identifying real vulnerabilities.
- **Low time overhead.** Many effective tools cannot be applied in industry, mainly because they require huge time cost to analyze binary programs. The low time overhead allows the vulnerability seeker to discover more known different vulnerabilities within a fixed period of time.

In this paper, we present *VulSeeker-Pro*, an enhanced semantic learning based vulnerability search tool which integrates function semantic emulation at the back end. For the function containing a vulnerability, it first uses the *semantic learning predictor* to quickly predict the top-M candidate functions from the target binary which are the most similar to the vulnerable function. Then the top-M candidate functions are input to the *emulation engine* to resort, and more accurate top-N candidate functions are obtained, where the value of N is much smaller than the value of M here. *VulSeeker-Pro* acquires a higher accuracy through the two-stage similarity score ranking process. Meanwhile, its time cost is basically the same as using semantic learning approach alone.

For evaluation, we compare *VulSeeker-Pro* with the state-of-the-art semantic learning based binary vulnerability search approach on 15 known vulnerabilities involving 6 widely used open source programs. The experimental results show that *VulSeeker-Pro* significantly outperforms the contrast tool *Gemini* [16] in terms of vulnerability search accuracy. In a total of 45 searches, *VulSeeker-Pro* ranks vulnerabilities 2nd on average in the target programs with a 24× ranking improvement, whereas *Gemini* ranks them 50th. In the top-1 and top-5 most similar results, *VulSeeker-Pro* found 40 and 43 real vulnerabilities, which are 12.33× and 2.58× more than *Gemini*. For *VulSeeker-Pro* and *Gemini*, it takes 1029.44 seconds and 849.40 seconds to complete a search task on the *OpenSSL* binary that contains 5995 functions. It is only 21.20% slower than *Gemini*. These demonstrate that *VulSeeker-Pro* is suitable for vulnerability search of large-scale code in industry.

2 RELATED WORK

Learning based vulnerability search. There have been several related works applying machine learning techniques to detect vulnerabilities in binary or source code. *Genius* [5] utilizes the spectral clustering to generate a codebook and calculates the similarity between a specific function and each representative function in the codebook based on the bipartite graph matching algorithm. *Gemini* [16] extracts the same lightweight instruction features as *Genius* and relies on the CFGs to generate the embedding vectors of the functions. Then the similarity of two embedding vectors is measured to get prediction result. *VulDeePecker* [8] uses code gadgets to represent programs and employs bidirectional LSTM neural network to automatically extract features instead of manually defining the features. These works have proven the effectiveness of machine learning techniques in vulnerability search area. However, due to the lack of sufficient and accurate semantic information, their precision is not very satisfactory.

Semantic computing based vulnerability search. Superior to syntax and structure information, semantic information can more accurately represent the program behavior. Furthermore, due to various compilation configurations provided by the compiler, the same source code may be compiled into various forms of binaries. So performing binary vulnerability search is actually dealing with the problem of semantic similarity detection, while syntax and structure based methods cannot well handle such a situation. *Bingo* [2] leverages selective inlining and length variant partial trace to compute function semantics, which constitutes function models to perform similarity comparison and vulnerability search. *BinSim* [11] calculates the equivalences of aligned system calls to better handle code obfuscation. It combines dynamic slicing with the weakest precondition calculation to identify fine-grained semantic similarities between two execution traces. *CACompare* [7] detects functionally similar code by extracting semantic signatures from the whole function while emulating the execution of a function. These work take semantic information into account to improve the search accuracy, but the semantic computing process imposes a significant time overhead.

Main Difference. Different from the above work, as far as we know, *VulSeeker-Pro* is the first tool that integrates function semantic emulation at the back end of semantic learning. It uses the fast predictive ability of a relatively accurate semantic learning approach to quickly pick out the top-M candidate functions which are most similar to the vulnerability from the target binary. Next, it performs function emulation on these top-M candidate functions to output more accurate top-N candidate functions. This combination not only reduces the time cost of dynamic approaches but also improves the search accuracy of semantic learning approaches, and release the engineers from the manual identification of true positive cases.

3 *VULSEEKER-PRO* DESIGN

The overall workflow of *VulSeeker-Pro* is shown in Figure 1. It contains two major modules: *semantic learning predictor* and *emulation engine*. The goal of *VulSeeker-Pro* is to determine whether the target binary contains functions similar to known vulnerabilities or not. Therefore, its input is a specific vulnerability and the target binary to be searched. *VulSeeker-Pro* utilizes the fast predictive capability of the semantic learning model to get the initial top-M (e.g., top-200) candidate functions by filtering out extremely dissimilar functions in the *semantic learning predictor* module. Then it resorts the top-M candidate functions based on dynamic execution traces of functions to generate the top-N (e.g., top-25) candidate functions as the final prediction results for the vulnerability in the *emulation engine* module.

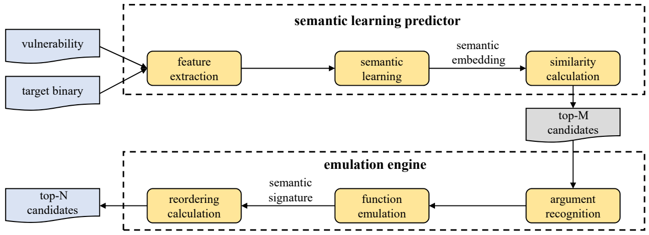


Figure 1: Overall workflow of *VulSeeker-Pro.*

3.1 Semantic Learning Predictor

The main goal of this module is to quickly filter out extremely dissimilar functions from the target binary and get the top-M (e.g., the value of M is 200) candidate functions that are most similar to a specific vulnerability. The key to the *semantic learning predictor* is to obtain the embedding vector representing the function semantics that can be used for similarity calculation. So *VulSeeker-Pro* consists of three main steps in this module.

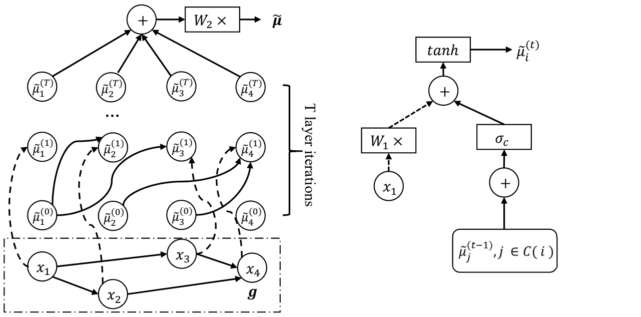
用IDA分析CFG，提取Genius提供的6个块和2个块为特征，编码为CFG为初始数字向量

3.1.1 Feature Extraction. The existing semantic learning approaches extract features for each basic block of the function and rely on the control flow graph (CFG) of the function to perform similarity comparison based on these extracted features[5, 16]. Similar to them, *VulSeeker-Pro* first disassembles the binary program into corresponding assembly program with the use of *IDA Pro* tool [12]. Then it uses *IDAPython* provided by *IDA Pro* [12] to create the CFG for each assembly function. Next, it extracts 6 intra-block features and 2 inter-block features provided by *Genius* [5] and encodes these 8 features into an initial numerical vector for each basic block of the CFG. The 8-dimensional initial numerical vector for each basic block of the function will be input to the next step to generate the semantic embedding vector of the entire function.

用Gemini的DNN网络训练

3.1.2 Semantic Learning. The main purpose of the semantic learning model is to obtain a high-dimensional (e.g, 64-dimensional) feature representation which can be used as the semantic embedding vector to represent the entire function semantics. After obtaining the 8-dimensional feature vectors for the specific vulnerability and each function in the target binary, two sets of feature vectors are fed to the semantic learning model to generate the semantic embedding vectors for similarity computation. *VulSeeker-Pro* regards the deep neural network (DNN) model of *Gemini* [16] as the semantic learning model. It is worth noting that any model with the ability to generate semantic embedding vectors can be considered as the semantic learning model. For example, our previous pure VulSeeker neural network [6] can also be integrated, and here we use *Gemini* for better and fairer comparison.

Figure 2(a) is a CFG denoted as $g = \langle V, E \rangle$, containing 4 vertices with initial numerical vectors: x_1, x_2, x_3, x_4, where V and E represent the vertex set and edge set, respectively. The DNN model contains a total of T (e.g., the value of T is 5) layer iterations, and each iteration will generate a new 64-dimensional embedding vector denoted as $\widetilde{\mu}_i^{(t)}$ of each vertex i. After obtaining the embedding vectors of all the basic block vertices within the function, *VulSeeker-Pro* aggregates them into the 64-dimensional embedding vector $\widetilde{\mu}$ of the function with the formula $\widetilde{\mu} = W_2(\Sigma_{i \in V} \widetilde{\mu}_i^{(T)})$, where W_2 is a 64×64 dimensional parameter matrix.


+
$W_2 \times$
$\widetilde{\mu}$
$\widetilde{\mu}_1^{(T)}$
$\widetilde{\mu}_2^{(T)}$
$\widetilde{\mu}_3^{(T)}$
$\widetilde{\mu}_4^{(T)}$
...
T layer iterations
$\widetilde{\mu}_1^{(1)}$
$\widetilde{\mu}_2^{(1)}$
$\widetilde{\mu}_3^{(1)}$
$\widetilde{\mu}_4^{(1)}$
$\widetilde{\mu}_1^{(0)}$
$\widetilde{\mu}_2^{(0)}$
$\widetilde{\mu}_3^{(0)}$
$\widetilde{\mu}_4^{(0)}$
x_1
x_2
x_3
x_4
g
$tanh$
$\widetilde{\mu}_i^{(t)}$
+
$W_1 \times$
σ_c
x_i
+
$\widetilde{\mu}_j^{(t-1)}, j \in C(i)$

(a) The DNN model for generating semantic embedding vector

(b) A schematic diagram for generating the embedding vector of each vertex during t-layer iteration.

Figure 2: The semantic learning model of *VulSeeker-Pro.*

Figure 2(b) illustrates the schematic diagram for generating the embedding vector $\widetilde{\mu}_i^{(t)}$ of each vertex i during the t-layer iteration. The input of the transformation process consists of two different parts: initial numerical vector x_i of the corresponding vertex i (the dotted arrow in Figure 2(a-b)), the sum $l_c^t = \Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}$ of previous embedding vectors of vertices pointing to vertex i (denoted as $C(i)$) through the CFG. The embedding vector of vertex i is calculated through the formula $\widetilde{\mu}_i^{(t)} = tanh(W_1 x_i + \sigma_c(l_c^t))$, where W_1 is an 8×64 dimensional parameter matrix. σ_c is a 2-layer fully-connected network represented as $\sigma_c(l_c^t) = P_1 \times ReLU(P_2 \times l_c^t)$, which is responsible for calculating an embedding vector with more powerful representation capability, where P_i is a 64×64 dimensional parameter matrix. We use *TensorFlow* [1] to implement the semantic learning model and apply the stochastic gradient descent algorithm to automatically learn model parameters, such as W_1,W_2,P_1 and P_2.

3.1.3 Similarity Calculation. Once obtaining the embedding vector $\widetilde{\mu}$ for target function and the embedding vector $\widetilde{\nu}$ for vulnerable function, *VulSeeker-Pro* calculates their similarity with the *Cosine* function $\widehat{y} = cos(\widetilde{\mu}, \widetilde{\nu}) = (\widetilde{\mu} \cdot \widetilde{\nu}) / (\|\widetilde{\mu}\| \cdot \|\widetilde{\nu}\|)$, where $\widehat{y}$ is the similarity score, ranging from -1 to 1. Based on the descending order of similarity scores between each function in the target binary and the vulnerable function, *VulSeeker-Pro* outputs the top-M candidate functions which are most similar to the vulnerability to the next module *emulation engine* for further processing.

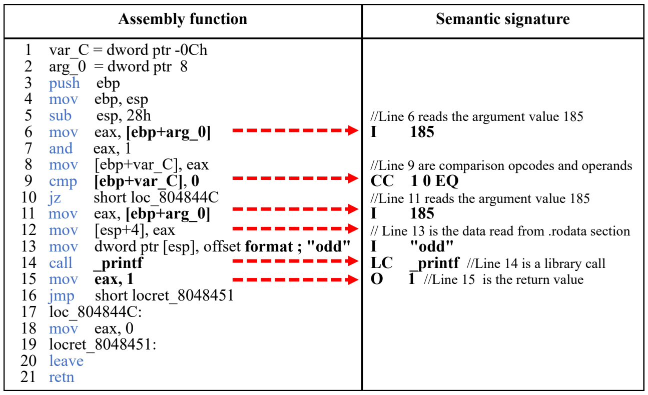
用cos计算两个函数的相似度

3.2 Emulation Engine

The main goal of this module is to reorder the input top-M candidate functions and get more accurate top-N (e.g., the value of N is 25) most similar candidate functions to the vulnerability. In order not to significantly increase the time cost of the vulnerability search, *VulSeeker-Pro* only performs function emulation for the top-M candidate functions with a low time cost to obtain semantic signatures. So it contains three main steps to achieve the reordering of candidate functions based on the semantic signatures representing the dynamic execution traces of functions.

3.2.1 Argument Recognition. Before emulating the execution of a function, *VulSeeker-Pro* first needs to recognize the arguments required for the function. The function arguments for the *X86* architecture binary include register arguments and stack arguments. The implementation of argument recognition is based on the assembly program disassembled from the binary program by *IDA Pro* [12]. The first three arguments of the function may be register arguments stored in the *EAX*, *EDX* and *ECX* registers. The *pseudocode* module of *IDA Pro* provides the ability to identify register arguments. The remaining arguments of the function are passed through the program stack whose space grows from high address to low address. Each function has a stack pointer indicating the stack start position corresponding to the function. When using *IDAPython* [12] to traverse assembly instructions of the function, if an instruction accesses a stack address that is larger than the stack start address, then the offset of the address relative to the stack start address is recorded as a stack argument.

3.2.2 Function Emulation. *VulSeeker-Pro* first generates a set of random integers for argument assignment before function emulation. For each function, the same random integer sequence is assigned to register arguments and stack arguments in turn. In addition, it uses *PyVEX* [14] to convert each assembly function into the *VEX-IR* (intermediate representation) [13] to perform function emulation in an easier way. Because *VEX-IR* statements have clear execution semantics, *VulSeeker-Pro* can emulate the execution of the function on the *VEX-IR* based on the assigned argument values. During the emulation execution of each function, it records the dynamic execution trace of the function called semantic signature. When *VulSeeker-Pro* encounters a call to function *B* while emulating function *A*, it also emulates 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 approach. Also if the emulation encounters an unknown memory space (*e.g.*, pointer reference), we assign a default value to the space.



Assembly function

```
1   var_C = dword ptr -0Ch
2   arg_0  = dword ptr  8
3   push   ebp
4   mov    ebp, esp
5   sub    esp, 28h
6   mov    eax, [ebp+arg_0]
7   and    eax, 1
8   mov    [ebp+var_C], eax
9   cmp    [ebp+var_C], 0
10  jz     short loc_804844C
11  mov    eax, [ebp+arg_0]
12  mov    [esp+4], eax
13  mov    dword ptr [esp], offset format ; "odd"
14  call   _printf
15  mov    eax, 1
16  jmp    short locret_8048451
17  loc_804844C:
18  mov    eax, 0
19  locret_8048451:
20  leave
21  retn
```

Semantic signature

```
//Line 6 reads the argument value 185
I     185

//Line 9 are comparison opcodes and operands
CC    1 0 EQ
//Line 11 reads the argument value 185
I     185
// Line 13 is the data read from .rodata section
I     "odd"
LC    _printf  //Line 14 is a library call
O     1  //Line 15  is the return value
```

Figure 3: The semantic signature recorded by *VulSeeker-Pro*.

The semantic signature of *VulSeeker-Pro* consists of four parts: *input values*, *output values*, *comparison opcodes/operands*, and *library function calls*. Figure 3 illustrates an assembly function and its semantic signature generated at the end of the emulation. Here the function argument is assigned to 185. *Input values* contain the data read from both the assigned argument values and the data sections (e.g., .rodata, .data). The instructions of lines 6, 11 and 13 in Figure 3 contain data reads, and their semantic information is marked "*I value*". *Output values* consist of the return value of the function and memory write values whose addresses are outside the range of the function stack. Line 15 in Figure 3 is the output value of the function when 185 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. Its example is line 9 in Figure 3, denoted as "*CC operands opcode*". *Library function calls* record the uses of *C* language standard library functions in the process of function emulation. Its semantic information is recorded as "*LC name*", such as line 14 in Figure 3.

3.2.3 Reordering Calculation. After obtaining semantic signatures of the vulnerable function and each function in the top-M candidate functions, *VulSeeker-Pro* uses the Jaccard similarity coefficient to calculate the similarity between them. The similarity score is calculated as follows: $J(A, B) = |A \cap B| / |A \cup B|$, where *A* and *B* are semantic signature sequences of the vulnerable function and the target function. By descending the similarity scores, *VulSeeker-Pro* reorders the top-M candidate functions and outputs more accurate top-N functions as the suspected vulnerabilities.

4 EXPERIMENTAL RESULTS

Experiment Setup. We conducted unbiased experiments for *Gemini* and *VulSeeker-Pro* in the same environment. All experiments were conducted on an 8-core 3.60GHz Intel i7 machine with 8G memory, an NVIDIA GeForce 1070 GPU and Ubuntu 14.0LTS. We apply the pre-training model of *Gemini* as the semantic learning predictor in *VulSeeker-Pro*. The training epoch for *Gemini* is 120, and other configurations are the same as the description in [16]. The values of M and N in *VulSeeker-Pro* are 200 and 25, respectively.

Table 1: Benchmark programs for vulnerability search

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

Benchmark Programs. To evaluate the accuracy and efficiency of *VulSeeker-Pro*, we select 15 known vulnerabilities from the *Common Vulnerabilities and Exposures* (CVE) website [3] as the benchmark for vulnerability search. As shown in Table 1, the benchmark consists of 6 widely used programs which involve a total of 15 randomly selected vulnerabilities at function granularity level. Column 1 gives the vulnerability number provided by the CVE organization. Columns 2 and 3 indicate which module the vulnerability exists in the corresponding program. Column 4 is the program version number affected by the vulnerability.

Each program is compiled into four optimization level versions (*O0-O3*) of the *X86* architecture using the GCC-4.8.4 compiler. We treat the vulnerable function in the *O3* version of the program as the source to search for it from the other three versions of the program. So we will perform 3 different searches for each vulnerability. In total, there are 45 different searches in the experimental evaluation. All the compiled programs and experimental results are available[1].

4.1 Accuracy Of Vulnerability Search

For each search of the vulnerable function, we obtain the top-200 candidate functions and the top-25 candidate functions from outputs of the *VulSeeker-Pro*. The top-200 candidate functions are *Gemini*'s rankings to the searched functions in the target program based on the descending order of similarity scores with the vulnerable function. And the top-25 candidate functions are the rankings of *VulSeeker-Pro* which aims at improving the predictive accuracy of the semantic learning approach (such as *Gemini*). Here we mainly focus on whether *VulSeeker-Pro* can identify vulnerabilities across compilation optimization options more accurately than *Gemini*.

In the experiment, we treat *O3* optimization level vulnerabilities as sources to perform search tasks from other three optimization level programs. Table 2 shows the search rankings of *Gemini* and *VulSeeker-Pro* on 15 vulnerabilities. Columns 2 and 6 are the search rankings for vulnerabilities in the *O0* optimization level programs by *Gemini* and *VulSeeker-Pro*, respectively. Columns 5 and 9 are the average rankings for vulnerabilities on the three optimization levels of *O0-O2*.

Table 2: The rankings of two tools on 15 vulnerabilities.

CVE No.	Gemini				VulSeeker-Pro			
	O0	O1	O2	Average	O0	O1	O2	Average
CVE-2018-11212	79	84	87	83	1	1	1	1
CVE-2018-11213	8	3	1	4	1	1	1	1
CVE-2018-11214	15	79	1	32	1	1	1	1
CVE-2018-0494	61	40	3	35	1	1	1	1
CVE-2017-6508	33	4	3	13	1	1	1	1
CVE-2015-1791	32	43	48	41	2	2	5	3
CVE-2014-3508	2	85	58	48	1	1	1	1
CVE-2016-6302	32	55	99	62	1	1	1	1
CVE-2016-6303	58	160	5	74	22	1	1	8
CVE-2016-2842	178	110	42	110	1	1	1	1
CVE-2014-9471	5	1	3	3	1	1	1	1
CVE-2017-7407	4	7	11	7	1	1	1	1
CVE-2015-3237	14	52	71	46	1	1	1	1
CVE-2015-3145	95	48	81	75	23	1	1	8
CVE-2015-3144	151	113	91	118	1	1	1	1
Average	51	59	41	50	4	1	1	2

From the last row of columns 5 and 9 in Table 2, we can see that *VulSeeker-Pro* ranks the vulnerable functions 2nd on average, whereas *Gemini* ranks 50th on average. In the top-1 candidate result among 45 searches, *Gemini* ranks only 3 vulnerabilities 1st, which results in a 6.67% accuracy. However, *VulSeeker-Pro* identifies 40 real vulnerabilities with an 88.89% accuracy, which is 12.33× higher than *Gemini*. Similarly, in the top-5 candidate functions, *VulSeeker-Pro* ranks 43 vulnerabilities 1st, leading to a 2.58× higher accuracy than *Gemini* (the value is 12). All vulnerabilities are ranked in the top-25 candidate functions by *VulSeeker-Pro*. In contrast, the value for *Gemini* is 17. On 12 vulnerabilities, *VulSeeker-Pro* consistently ranks the vulnerabilities 1st in the three optimization levels of programs. No vulnerability *Gemini* can always rank it 1st.

By counting the rankings from column 2 to column 4, we find that 5 out of 45 searches, *Gemini* ranks the vulnerabilities behind 100. We look at the assembly functions of the corresponding vulnerability and summarize two reasons that can lead to this situation. One is that function inlining exists in the higher optimization level binary functions during compiling, which affects the instruction features of function. The other reason is that the CFGs of the same function are changeable under different optimization configurations, which is reflected in the semantic embedding vector of the function. These two reasons lead to inaccurate prediction results. For *VulSeeker-Pro*, two vulnerabilities are ranked outside the top-10 candidate results in *O0* optimization level searches. This is because constant integers in programs are treated as memory references to constant data sections (e.g., .rodata section) in the *O0* optimization level, and is directly used as immediate values in the *O3* optimization level. It means that the compiler can use different instruction addressing modes to accomplish the same purpose at different optimization levels. Due to this situation is common, it affects semantic signatures of functions to a certain extent, so *VulSeeker-Pro* achieves a poor ranking in rare cases.

In summary, *VulSeeker-Pro* significantly outperforms *Gemini* with a 24× ranking improvement on average in terms of vulnerability search accuracy. This improvement consists in the integration of function emulation at the back end of the semantic learning approach to generate accurate function semantics.

4.2 Time Efficiency

We evaluate how much time *VulSeeker-Pro* needs to complete a vulnerability search on 6 open source programs. This will have a direct bearing on whether *VulSeeker-Pro* can be used in industry. Table 3 shows the time cost of the two tools for each program to which vulnerabilities belong. Column 2 lists the number of functions in each program. Columns 3 and 4 are the time for *Gemini* and *VulSeeker-Pro* to complete a vulnerability search.

Table 3: Time cost of the two tools on each program.

Program	#Functions	Gemini/s	VulSeeker-Pro/s
libjpeg	580	81.20	176.45
Wget	804	116.28	241.34
OpenSSL	5995	849.40	1029.44
Coreutils	119	19.78	38.41
curl-7.53.1	1113	166.95	306.84
curl-7.40.0	2760	469.20	659.35
AVG	**1895**	**283.80**	**408.64**

In this experiment, each program contains an average of 1895 functions. It spends 283.80 seconds and 408.64 seconds on average respectively for *Gemini* and *VulSeeker-Pro* in completing a vulnerability search. *Gemini* and *VulSeeker-Pro* need an average of 0.15 seconds and 0.22 seconds to calculate the similarity between a target function and a vulnerable function. Looking closely at columns 2 through 4 of Table 3, we find that the time cost of *Gemini* increases linearly with the number of functions roughly. However, the time cost of *VulSeeker-Pro* violates this phenomenon. The main reason is that *Gemini* needs to extract features, generate semantic embeddings, and compute similarity to the vulnerability for all functions of the program. But *VulSeeker-Pro* only needs to perform the emulation for the top-200 candidate functions. Therefore, the more the number of functions is in the program, the closer the time spent by *VulSeeker-Pro* is to *Gemini*.

[1] They are available at https://github.com/buptsseGJ/VulSeeker-Pro

In summary, for a single function, although the time cost of *VulSeeker-Pro* is 0.07 seconds more than *Gemini* on average, we can always achieve a higher search accuracy in a reasonable time. This is valuable to reduce the amount of time needed to manually confirm lots of fake vulnerabilities by adding a little time for more accurate vulnerability predictions.

5 DISCUSSION

In the experimental evaluation, we have demonstrated the high efficiency and accuracy of *VulSeeker-Pro* in vulnerability search. However, we also identify some clone-based search limitations and related requirements in industry and discuss some possible solutions to satisfy those requirements.

Engineers Prefer Tools Without Manually Identification. By integrating function semantic emulation on the back end of the semantic learning approach, we greatly improve the prediction effectiveness of the vulnerability and reduce the human efforts to identify the true positive vulnerable functions from the top-N search results. Engineers are positive and are willing to apply this automatic support even with about 21.20% time overhead. Furthermore, the performance of this approach depends on the top-M candidate functions output by the semantic learning approach. If we can reduce the value of M without losing accuracy, the total vulnerability search time required by *VulSeeker-Pro* will be shortened. We list several ways to increase efficiency as follows:

- When training the deep neural network (DNN) model in the semantic learning approach, we can enhance the generalization ability of the model by increasing the discrete training samples from multiple binary programs. In general, the DNN model with a larger training epoch will have stronger vulnerability prediction ability. However, we need to pay attention to preventing model over-fitting.
- The data flow graph (DFG) of the function depicts the transfer and use of data within the function. Including both CFG and DFG in function semantic embedding vector generation will effectively mitigate the negative impact of the CFG structure changing at different optimization levels on the prediction results.

High Accuracy Cross-Platform Search Is Greatly Needed. With the popularity of terminal devices, software programs on traditional *X86* architecture are gradually being compiled and ported to other architectures (e.g., *ARM*, *MIPS*). Therefore, the proposed tool should also support cross-architecture vulnerability search and still remain a high search accuracy. Since the approach involved in *VulSeeker-Pro* are universal, this goal is not difficult to achieve.

VulSeeker-Pro contains two main modules, for which we need to make appropriate extensions. For the *semantic learning predictor* module, we only need to provide cross-architecture support in the feature extraction part. The 8 types of features used in *VulSeeker-Pro* need not be changed, but the instructions contained in each feature need to follow the corresponding architecture. This extension can be easily accomplished by referring to the software developer's manual of a specific architecture. For the *emulation engine* module, since we execute function emulation on an architecture-independent intermediate representation (specifically, *VEX-IR*), the main content of this module need not be changed. We only need to extend the argument recognition part slightly. The number and naming of register arguments used in different architectures are significantly different. For example, *EAX*, *EDX* and *ECX* are register arguments for the *X86* architecture. However, they are *R0*, *R1*, *R2* and *R3* for the *ARM* architecture. We need to identify register arguments and stack arguments following the calling conventions of the specific architecture before emulating functions.

6 CONCLUSION

In this paper, we present *VulSeeker-Pro*, a binary vulnerability seeker that integrates function emulation at the back end of semantic learning. With fast filtering of semantic learning and dynamic trace generation of function emulation, we can achieve more accurate search results with low time overhead. Experimental results show that *VulSeeker-Pro* ranks vulnerabilities in the target programs at the 2nd on average with a 24× ranking improvement than the most recent and related work *Gemini*. In a total of 45 searches, *VulSeeker-Pro* finds 40 and 43 real vulnerabilities in the top-1 and top-5 candidate functions, which are 12.33× and 2.58× more than *Gemini*. Performing a vulnerability search task on the *OpenSSL* binary containing 5995 functions takes only 1029.44 seconds with an average of 0.17 seconds per function. For all of the 6 programs with 11,371 functions, *VulSeeker-Pro* needs an average of 0.22s to calculate the similarity between a target function and a vulnerable function, which is only 0.07 seconds slower than *Gemini*. These demonstrate that *VulSeeker-Pro* is suitable for vulnerability search of large-scale code in industry.

REFERENCES

[1] Martın Abadi, Ashish Agarwal, Paul Barham, et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture cross-os binary search. In *FSE*. ACM, 678–689.

[3] CVE. [n. d.]. Common Vulnerabilities and Exposures. http://cve.mitre.org/. Accessed June 15, 2018.

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

[5] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *CCS*.

[6] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: a semantic learning based vulnerability seeker for cross-platform binary. In *ASE*.

[7] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *ICPC*. IEEE.

[8] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *CoRR* abs/1801.01681 (2018).

[9] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *25th International Conference on Software Analysis, Evolution and Reengineering*.

[10] 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*.

[11] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based semantic binary diffing via system call sliced segment equivalence checking. USENIX.

[12] IDA Pro. [n. d.]. The IDA Pro Disassembler and Debugger. https://www.hex-rays.com/. Accessed May 20, 2018.

[13] Julian Seward and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*. 89–100.

[14] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *NDSS*.

[15] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. In *ICSE*.

[16] 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*. ACM, 363–376.