VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary

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

Jiaguang Sun
School of Software, Tsinghua University
Beijing, China

ABSTRACT

Code reuse improves software development efficiency, however, vulnerabilities can be introduced inadvertently. Many existing works compute the code similarity based on CFGs to determine whether a binary function contains a known vulnerability. Unfortunately, their performance in cross-platform binary search is challenged.

This paper presents *VulSeeker*, a semantic learning based vulnerability seeker for cross-platform binary. Given a target function and a vulnerable function, *VulSeeker* first constructs the labeled semantic flow graphs and extracts basic block features as numerical vectors for both of them. Then the embedding vector of the whole binary function is generated by feeding the numerical vectors of basic blocks to the customized semantics aware DNN model. Finally, the similarity of the two binary functions is measured based on the Cosine distance. The experimental results show that *VulSeeker* outperforms the state-of-the-art approaches in terms of accuracy. For example, compared to the most recent and related work Gemini, *VulSeeker* finds 50.00% more vulnerabilities in the top-10 candidates and 13.89% more in the top-50 candidates, and improves the values of AUC and ACC for 8.23% and 12.14% respectively. The video is presented at https://youtu.be/Mw0mr84gpI8.

CCS CONCEPTS

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

KEYWORDS

semantic learning, vulnerability search, cross-platform binary

ACM Reference Format:
Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240480

*Also with Beijing National Research Center for Information Science and Technology.
†Correspondence author

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240480

1 INTRODUCTION

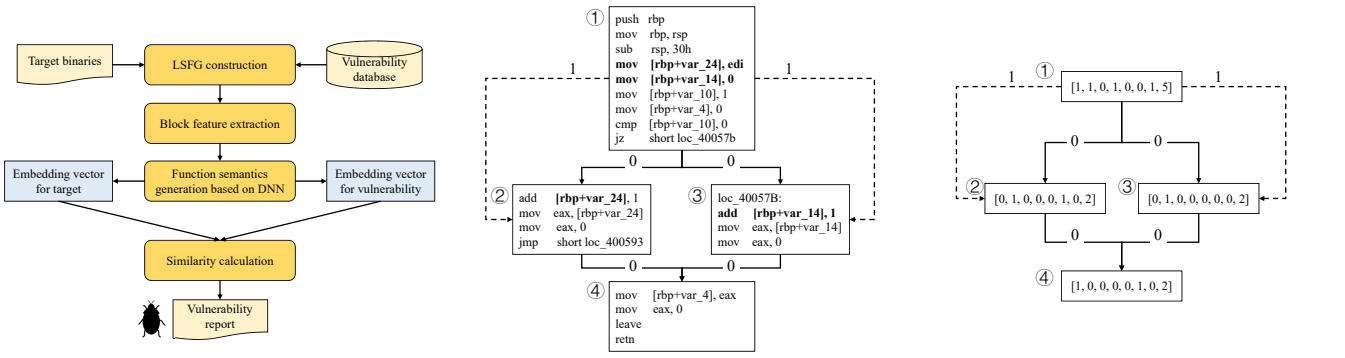
In order to improve the efficiency of software development, copy-paste of code and reuse of third-party libraries are common. If such a development process is poorly managed, unpatched vulnerable code can easily be propagated to different software programs. With the popularity of terminal devices, software programs on traditional x86 architecture are gradually being compiled and ported to other architectures (e.g., ARM, MIPS). As a consequence, more and more binary programs contain a large number of similar or identical vulnerable codes. For example, 145 unpatched clone vulnerabilities are confirmed in the Debian system [7].

Many works have been presented to perform vulnerability search for cross-platform binaries based on code clone techniques [3, 5, 6, 9, 14] or fuzzy testing [8, 13]. Most of the clone-based techniques analyze the control flow graph (CFG) to determine whether a binary contains vulnerabilities or not [3, 5, 6, 14]. They perform well on their settings, but the accuracy and efficiency may lose coming across to large-scale binaries.

In this paper, we present *VulSeeker*[1], a semantic learning based vulnerability seeker for cross-platform binary. *VulSeeker* acquires a higher accuracy and efficiency through the labeled semantic flow graph (LSFG) construction and the semantics aware deep neural network (DNN) based function semantics generation. The LSFG contains both the CFG and DFG (data flow graph), so more semantic information of a binary function is captured than using the CFG alone. The semantics aware DNN model transforms numerical features of basic blocks within the function into function semantics (or embedding vector). Vulnerability is identified by measuring the similarity of two binary functions based on the *Cosine* distance of their embedding vectors.

For evaluation, we compare *VulSeeker* with the state-of-the-art cross-platform binary clone vulnerability search approach on some widely used third-party benchmarks consisting of real-world applications. The experimental results show that *VulSeeker* outperforms the most recent and related work Gemini [14]. On average, in terms of clone detection, the AUC and ACC of *VulSeeker* are 88.49% and 81.3%, which are 8.23% and 12.14% higher than those of Gemini. Furthermore, we use the CVE-2015-1791 vulnerability to evaluate the vulnerability search capability in 4643 firmware images. In the top-10 and top-50 most similar results, *VulSeeker* found 50.00% and 13.89% more real vulnerabilities than Gemini.

[1] *VulSeeker* is available at https://github.com/buptsseGJ/VulSeeker



(a) Overall architecture and workflow of *VulSeeker*. (b) An example of the LSFG construction. (c) An example of the block feature extraction.

Figure 1: *VulSeeker* design. (a) is the overall architecture. (b-c) are examples for LSFG construction and block feature extraction.

2 RELATED WORK

Binary Clone Detection. *COP* [9] is a plagiarism detection tool that combines program semantics with longest common subsequence based fuzzy matching. *BinGold* [2] extracts the semantics of binary code in terms of both the DFG and the CFG, and synthesizes them into a novel representation called the semantic flow graph. However, it does not support cross-architecture clone detection. *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.

Vulnerability Search. *Bingo* [3] leverages selective inlining and length variant partial trace to compute function semantics, which constitute function models to perform similarity comparison and vulnerability search. *Genius* [6] utilizes the spectral clustering to generate a codebook and calculates the similarity between a specific ACFG and each representative ACFG in the codebook based on the bipartite graph matching algorithm. *Gemini* [14] extracts the same lightweight features as *Genius* and only relies on the CFG to generate the embedding vector of the function. Then the similarity of two embedding vectors is measured to get a prediction result.

Main Difference. Different from the above work, as far as we know, *VulSeeker* is the first tool that combines CFG and DFG to form LSFG, and applies deep learning to perform vulnerability search for cross-platform binary. It extracts 8 types of lightweight instruction features for each basic block in LSFG. Based on the graph topology and the revised semantics aware DNN model, we apply 6 layer iterations to the LSFG to obtain the semantic representation of the entire binary function, and acquire higher accuracy.

3 *VULSEEKER* DESIGN

The overall workflow of *VulSeeker* is shown in Figure 1(a). It contains four major components: *LSFG construction*, *block feature extraction*, *function semantics generation* and *similarity calculation*. The goal of *VulSeeker* is to determine whether the target binary contains functions similar to known vulnerabilities or not. Therefore, its input is two binary functions from the target binary and the vulnerability database. Firstly, *VulSeeker* constructs the LSFGs for the two binary functions. Then it extracts 8 types of lightweight instruction features and encodes them as a numerical vector for each basic block of the LSFG. Function semantics is generated by feeding the numerical vectors of basic blocks within the LSFG to the semantics aware deep neural network (DNN) model. Finally, *VulSeeker* outputs whether the target binary function contains a known vulnerability or not based on the similarity of embedding vectors of the two input functions.

3.1 LSFG Construction

Labeled semantic flow graph contains both the CFG and the data flow graph (DFG), and their edges are marked as 0 and 1 respectively. Its purpose is to improve the accuracy of function semantics generation, because it considers both the control structure and the data transfer within a function, which will effectively mitigate the structural interference introduced by the varying CFG under different platforms. Figure 1(b) illustrates an example of the LSFG.

We use *IDAPython* provided by *IDA Pro* [12] to create the CFG for the basic blocks of each binary function. Based on the CFG, we infer whether there should be a data pointing edge between two basic blocks by leveraging the *LLVM IR* plugin [10] on *IDA Pro*. For two instructions i and j from two different basic blocks which meet the CFG topology, if the instruction i writes a memory location and the instruction j reads the same memory address, we create a data dependent edge for these two blocks. In addition, only the data dependencies between different blocks are preserved, and there is at most one data dependent edge between two basic blocks. *VulSeeker* stores the control edges and data edges of each function in two files.

3.2 Block Feature Extraction

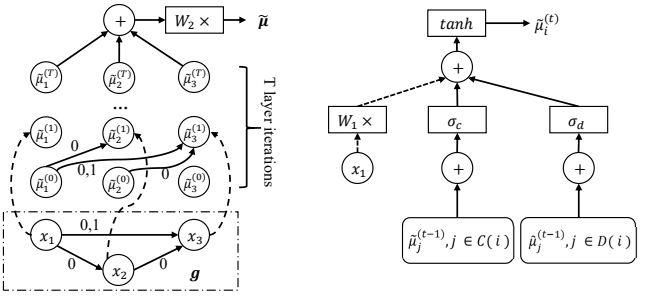
By referring to features used in previous works [2, 6] and executing a series of code clone experiments for different feature sets, we have finally determined to use 8 types of features shown in Table 1 as the initial semantic representation of each basic block. These selected features are lightweight and robust, which can be easily extracted and change little under various implementation platforms with different microprocessor architectures and various compilation optimization configurations. We utilize the *IDAPython* to extract features for each basic block. Then we encode the 8 features of each basic block as a numerical vector. Figure 1(c) is the numerical vectors of each basic block corresponding to the function in Figure 1(b). For each binary function, numerical vectors of all the basic blocks within the function are stored in a separate file.

Table 1: Basic-block level features used by *VulSeeker*.

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

3.3 Function Semantics Generation

The input of this component is the d dimensional initial numerical vectors of all the basic blocks within the function, and the output is the p dimensional embedding vector representing the function semantics. To precisely capture function semantics, data and control dependencies between basic blocks along the LSFG topology need to be considered. Referring to the *structure2vec* neural network [4], we propose a semantics aware DNN model shown in Figure 2 that specializes in processing structured LSFG representation.



(a) The DNN model for generating function semantics (or embedding vector)

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

Figure 2: The DNN model of *VulSeeker*.

Figure 2(a) is a LSFG denoted as $g = \langle V, E \rangle$, containing three vertices with initial numerical vectors: x_1, x_2, x_3, where V and E represent the vertex set and edge set, respectively. Edges marked 1 and 0 indicate data dependency and control dependency, respectively. The DNN model contains a total of T layer iterations, and each iteration transforms the initial numerical vector x_i of each vertex i into its embedding vector $\widetilde{\mu}_i^{(t)}$. After obtaining the embedding vectors of all the basic block vertices within the function, we aggregate them into the p 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 $p \times p$ dimensional parameter matrix.

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 three different parts: initial numerical vector x_i of the corresponding vertex i (the dotted arrow in Figure 2(a-b)), the sum of previous embedding vectors of vertices pointing to vertex i through the control dependency (denoted as $C(i)$), and the sum of previous embedding vectors of vertices pointing to vertex i through the data dependency (denoted as $D(i)$). The embedding vector of vertex i is calculated through the formula $\widetilde{\mu}_i^{(t)} = tanh(W_1 x_i + \sigma_c(\Sigma_{j \in C(i)} \widetilde{\mu}_j^{(t-1)}) + \sigma_d(\Sigma_{j \in D(i)} \widetilde{\mu}_j^{(t-1)}))$, where W_1 is a $d \times p$ dimensional parameter matrix. σ_c and σ_d are two n layer fully-connected networks responsible for calculating an embedding vector with more powerful representation capability, they are represented as follows:

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

where n is the embedding depth of each vertex, P_i and Q_i are $p \times p$ dimensional parameter matrixes. Through T layer iterations, the feature of each vertex is propagated to other vertices as the iteration progresses along with the LSFG topology, ensuring that each basic block of the function has corresponding context semantics.

3.4 Similarity Calculation

Once obtaining the embedding vector $\widetilde{\mu}$ for target function and the embedding vector $\widetilde{\nu}$ for vulnerable function, *VulSeeker* calculates their similarity with the *Cosine* function $\widehat{y} = cos(\widetilde{\mu}, \widetilde{\nu}) = \frac{\widetilde{\mu} \cdot \widetilde{\nu}}{\|\widetilde{\mu}\| \|\widetilde{\nu}\|}$, where $\widehat{y}$ is the similarity score, ranging from -1 to 1. If the similarity score $\widehat{y}$ is larger than a pre-defined threshold, the target binary function is considered similar to the vulnerability. We use *TensorFlow* [1] to implement the semantics aware DNN model and apply the stochastic gradient descent algorithm to automatically learn model parameters, such as W_1,W_2,P_1 and Q_1.

4 EXPERIMENTAL RESULTS

VulSeeker mainly contains 3 executable files that can be used based on the following steps: 1) modify the *config.py* file to configure the target programs for vulnerability search; 2) execute the *command.py* file to generate the LSFGs and extract initial numerical features for basic blocks; 3) execute the *search_by_list_vulseeker.py* file to obtain the embedding vectors of functions and get the function list in descending order of similarity scores.

VulSeeker is evaluated on two datasets. Dataset I contains 735,540 functions with 9,345K basic blocks. We compile *BusyBox* (v1.21.0), *OpenSSL* (v1.0.1f and v1.0.1u) and *Coreutils* (v6.5 and v6.7) in *X86, X64, MIPS32, MIPS64, ARM32, ARM64* architectures, using *GCC* (v4.9 and v5.5) with optimization levels O0-O3. Dataset II consists of 4643 firmware images for various architectures from [6]. All experiments are conducted on default configurations of the DNN model as follows: the embedding depth n is 2, the embedding size p is 64, the number of iterations T for each basic block is 6, the training epoch is 100.

Accuracy of Code Clone. We treat two different compiled versions of the same source function as a pair of similar functions and vice versa. We randomly select 2500 pairs of similar functions and 2500 pairs of dissimilar functions from the dataset I to perform the comparative experiment with Gemini configured with the optimal parameters [14]. Figure 3 is their ROC (receiver operating characteristic) curves of code clone. We observe that the ROC curve of *VulSeeker* is above Gemini, which means that *VulSeeker* can achieve a higher true positive rate at the same false positive rate. The AUC value and ACC value of *VulSeeker* are 88.49% and 81.3%, which are 8.23% and 12.14% higher than those of Gemini. In summary, *VulSeeker* outperforms Gemini, because in addition to the CFG, we also construct the DFG for tracking the usages of variables between basic blocks. During function semantics generation, we obtain more robust semantic information with a revised DNN structure, which is beneficial to the effective identification of clone functions.

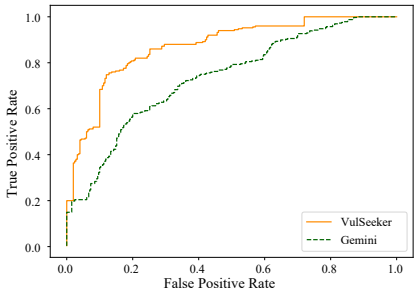


Figure 3: ROC curves of code clone.

Furthermore, the experimental result of Gemini here is lower compared with the description of the literature [14]. Two reasons lead to this situation. One is that our dataset I contains 5 programs, but Gemini only contains 2 of them. The other is that we compile these programs into six architectures, including three 64-bit ones, and Gemini only compiles them into three 32-bit ones. The number of general-purpose registers used in 32-bit and 64-bit architectures is different, which affects the feature vectors of basic blocks, and its results drop accordingly. In their relatively simpler setting, they could acquire the AUC value of 97%, and we could improve the value to almost 99%. We found that the more complex the dataset is, the more improvements *VulSeeker* would achieve, and the performance of Gemini drops faster than *VulSeeker* for complex settings.

Accuracy of Vulnerability Search. We employ dataset II to evaluate the effectiveness of *VulSeeker* and Gemini in vulnerability search. We take *CVE-2015-1791* with 48 compiled versions as the vulnerable function. For each version of the vulnerability, we employ the two tools to perform the search task from known vulnerable firmware images. For each firmware image, we sort the functions in descending order of the average similarity scores for 48 searches. *VulSeeker* ranks the vulnerability function 8th on average, whereas Gemini ranks 99th on average. If we take the highest rankings out of 48 searches for each firmware image, *VulSeeker* has a 100% chance of finding the vulnerable function in top-3 candidates which is 11.76% higher than Gemini.

Table 2: The accuracy of vulnerability search

top-K	Gemini		VulSeeker	
	#Num	Percent	#Num	Percent
1	1	100%	1	100%
5	2	40%	3	60%
10	4	40%	6	60%
50	36	72%	41	82%
100	75	75%	83	83%

For the search results of the *MIPS32* version vulnerability, we sort the functions in all firmware images in descending order of similarity scores. Table 2 shows the effectiveness of vulnerability search on the top-*K* most similar results among all functions. Column 1 is the different *K* value. Columns 2 and 4 are the number of real vulnerabilities in the top-*K* results, and columns 3 and 5 are the percentage of corresponding real vulnerabilities. We can see that *VulSeeker* has a great improvement on the search precision than Gemini. In top-10 results, *VulSeeker* finds 50% more vulnerabilities than Gemini. In summary, *VulSeeker* outperforms Gemini in the vulnerability search in terms of the CVE-2015-1791. We also do some evaluation on other CVEs, and the performance improvements remain.

Time Cost. *VulSeeker* mainly consists of four components, and for the size of the experimental database, it can determine whether the given binary function contains a known vulnerability within an average of 0.20 seconds, while Gemini takes about 0.15 seconds. Function semantics generation takes up almost half of the time cost, and its time cost grows linearly with the number of basic blocks within the function. Although the time cost of *VulSeeker* is 0.05 seconds more than Gemini, we can achieve a higher search accuracy in a reasonable time.

5 CONCLUSION

In this paper, we present *VulSeeker*, a cross-platform binary vulnerability seeker based on semantic learning. With integrating the CFG and the DFG of the binary function, we capture more function semantics. Experimental results show that *VulSeeker* achieves 88.49% AUC value and 81.3% ACC value for code clone, which improves 8.23% and 12.14% than Gemini, respectively. In the case study of CVE-2015-1791 vulnerability search, *VulSeeker* finds 50.00% more vulnerabilities in the top-10 candidates and 13.89% more in the top-50 candidates. For the time cost of vulnerability search, *VulSeeker* needs 0.20 seconds to determine whether a function has a known vulnerability or not in the relatively huge database. These demonstrate that *VulSeeker* is suitable for vulnerability search of large-scale code.

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] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. 2016. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Digital Investigation* 18 (2016), S11–S22.
[3] 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 *SIGSOFT FSE*. ACM, 678–689.
[4] Hanjun Dai, Bo Dai, and Le Song. 2016. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning*. 2702–2711.
[5] Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical similarity of binaries. *ACM SIGPLAN Notices* 51, 6 (2016), 266–280.
[6] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable graph-based bug search for firmware images. In *CCS*. ACM, 480–491.
[7] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 48–62.
[8] 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*.
[9] 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 *SIGSOFT FSE*.
[10] MIASM. [n. d.]. Reverse Engineering Framework. https://github.com/cea-sec/miasm. Accessed May 20, 2018.
[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] 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*.
[14] 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.