SemHunt: Identifying Vulnerability Type with Double Validation in Binary Code

Yao Li, Weiyang Xu, Yong Tang, Xianya Mi, and Baosheng Wang
College of Computer Science
National University of Defense Technology
Changsha, Hunan, China
{liyao15,xuweiyang11,ytang,mixianya09}@nudt.edu.cn,wangbaosheng@126.com

***Abstract*—when manufacturers release patches, they are usually released as binary executable programs. Vendors generally do not disclose the exact location of the vulnerabilities, even they may conceal some of the vulnerabilities, which is not conducive to study the in-depth situation of security for the need of consumers. In this paper we introduce a vulnerability discover method using machine learning based on patch information - SemHunt. Firstly, we use it to compare two versions of the same program to get the potential vulnerability-patched function pairs using binary comparison technology. Then, we combine it with vulnerability and patch knowledge database to classify these function pairs and identify the possible vulnerable functions and the vulnerability types. We completed a prototype of SemHunt, which can effectively identify vulnerable function types and the location of corresponding vulnerabilities, which are not revealed in the released patch files. Finally, we test some programs containing real-world CWE vulnerabilities, and one of the experimental results about CWE843 shows that the results returned from only searching source program are about twice as much as the results from SemHunt. We can see that using SemHunt can significantly reduce false positive rate of discovering vulnerabilities compared with analyzing source files alone.**

***Index Terms*—software security; binary comparison; vulnerability; patch file; machine learning**

I. Introduction

At present, the search for vulnerable software functions is mainly based on software source code, which means that professional testing tools on source code can find the vulnerabilities through automatic analysis of the code. However, obtaining the source code is a very demanding condition, because many software only provides executable programs such as the large commercial software MS Office and the free off-source software Adobe Reader. In addition, manufactures usually release patches in binary executable program without detailed vulnerability information and its corresponding location. Therefore, finding vulnerabilities in the source code has great limitations. They are not conducive to study the in-depth situation of security for the need of consumers. Now, there are some problems in vulnerability analysis techniques at the binary level. Firstly, the binary comparison technique between two versions of the same program may miss some match functions[1][2][3][4], and the analysis of the vulnerability still needs a lot of manual work. There are some binary comparison tools like BinDiff[5][6], BinSlayer[7] and so on[8], which can get good matching results. They compare two functions to get their similarity, and determine whether the two functions match by setting a threshold. In contrast, our method aims at finding the potential vulnerability-patched function pairs, which requires us to modify the current binary comparison algorithm to adapt to our method.

DOI reference number: 10.18293/SEKE2017-117

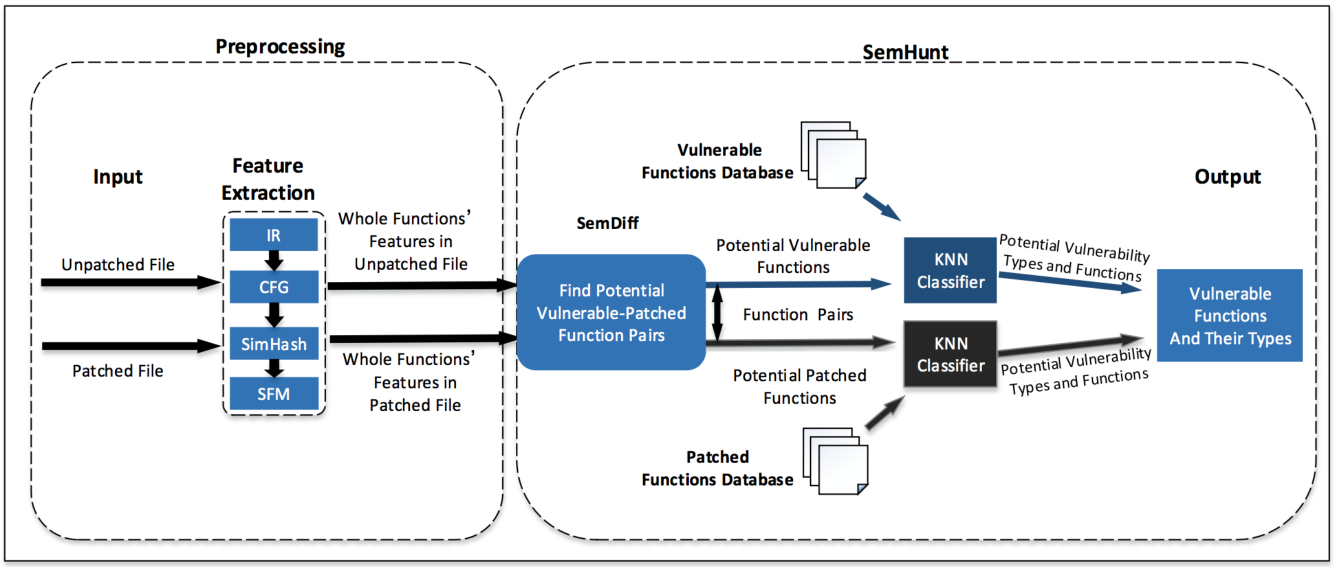
Another problem is that the search for vulnerability at the source program level may have high false positive rate[9]. They establish the vulnerability knowledge database and find the similar functions to the test ones to determine the vulnerability types. Eschweiler and etc. match the function basic blocks' flow graph to find the pairing vulnerable functions to the test software function[10]; Ming and etc. firstly find the match basic blocks through semantic features, then extend the matched blocks to the functions[1]. These methods only identify the possible vulnerable functions and their vulnerability types at the source program level. Lacking verification from other ways may lead to the high false positive rate.

Our paper firstly presents a novel approach to finding the semantic differences between two versions of the same program, aiming at finding the exact match between the potential vulnerable functions and potential patched functions. Then, we use the machine learning classification algorithms which are combined with the patch information to make a double verification for identifying vulnerability types. Thus, we can increase the accuracy of the vulnerability report, as well as reduce the false positive rate and identify these vulnerable functions types and corresponding locations more effectively, which are not disclosed in detailed information when the patch file is released.

The main contributions of the paper are as follows:

(i) We build a set of common vulnerability functions, as well as a corresponding set of patch-functions, which are used to train the machine-learning classifier.
(ii) We propose a new binary comparison method between the unpatched-vision and patched-vision software, to find out the possible vulnerability-functions and its patch functions.
(iii) We have implemented a prototype of SemHunt.

Our paper will be described in the following order. Section 2 will introduce a whole framework. Section 3 will introduce the key points of the SemHunt. Section 4 will introduce



Preprocessing
SemHunt
Input
Feature Extraction
Unpatched File
Patched File
IR
CFG
SimHash
SFM
Whole Functions' Features in Unpatched File
Whole Functions' Features in Patched File
SemDiff
Find Potential Vulnerable-Patched Function Pairs
Vulnerable Functions Database
Potential Vulnerable Functions
Function Pairs
Potential Patched Functions
Patched Functions Database
KNN Classifier
KNN Classifier
Potential Vulnerability Types and Functions
Potential Vulnerability Types and Functions
Output
Vulnerable Functions And Their Types

Fig. 1: System Architecture

the experience in detail. Weakness and future work will be introduced in Section 5.

II. System Architecture

Figure 1 shows our overall system structure. Firstly, the two binary files are loaded into our angr platform, and it run its own disassembly tool to generate the assembly code. Then the assembly code are transferred into the intermediate language using the IR converter(VEX), and the CFG is generated through the CFG constructor. We extract the semantics information and digital feature of the function. After that we use them in the binary comparison algorithm to get the possible pairs of unpatched-function and patched-function, we extract the feature and Simhash of each function through the digital feature extractor. We add the match between patched-file and patched-function set to reduce the rate, because only finding the match function between vulnerability-functions set and the unpatched file to get the vulnerability type may have a high false positive rate. Then we put the features into the machine learning classifier to get the corresponding matches. We combine the results from two classifier to get the final potential vulnerability types. Then we will talk about some parts briefly.

III. The Keys Of SemHunt

There are two main parts in the SemHunt: Binary comparison algorithm and Vulnerability Search Combined With Patch Information.

A. Binary comparison algorithm

We modify the BinDiff algorithm to adapt our system. Bindiffs selectors only consider the structural characteristics of the three-tuple information, such as its selector of callgraph, which only includes the numbers of basic blocks, the number of edges and a sub-function call number. The selector for the control flow graph of the basic block contains the number of blocks of the shortest path to the exit of the function, the number of blocks passed from the entry point to the shortest path of the block and the number of points of the sub-function in the basic block. They are all used in a Euclidean space to carry out the minimum distance to judge[5].

However, our SemDiff changes not only in the selector, including the semantic judgments and more digital information, but also in the process of loop. We do not only check the parent-child nodes in one loop (this method may miss a lot of matches), but also carry out their own check. To determine the two functions are functionally identical, we define a rule as follows:

Definition 1: pair functions equivalence formulas of data-Given two list of data that are recorded in the memories and registers: $X = [x_0, x_1, \cdots, x_m]$, and $Y = [y_0, y_1, \cdots, y_n]$. For every x_i in X, there is a bijection, $y_j = f(x_i)$ is existed ,then ,we call the data is same. The formula is described as follows:

$$\forall x_i, \exists y_j = f(x_i), f(x)\ is\ a\ bijection$$

There are three parts in the SemDiff: WholeMatch, MatchPro and SemDiff. As with the previous algorithms, at the beginning of the initially matching process, we find the functions which are uniquely matched, which means that, we find some functions that have the same symbol expressions and digital features.

At the first stage of the algorithm, the inputs are the function sequences of the two programs respecitively. After the ”Whole Match”, we get a result of the match function and two lists of functions that are still unmatched.

Then, it is the second stage. We call it ”MatchPro”, because it is applied in the propagation progress. The input of the second part is the matched function set, which is got from the first stage and the remaining unmatched function sequences . Then we match each function from a small set, which can be got after the function ”FindPaAndCh”. It is a function that return the set of the parent nodes and children nodes to the functions in the matched set. After we get the subset, we use them into the ”WholeMatch” to get the matched functions until we finish traversing all matched functions.

The ”SemDiff” is a combination of ”WholeMatch” and ”MatchPro”. After we execute the three process, we get a matched-function set and an unmatched-function set. The match set includes the functions that are absolutely the same, so they are not the vulnerable functions and corresponding patched functions. We can get the potential vulnerable-patched function pairs in the unmatched-function set. We compare each function pair in it. If the similarity is over 80%, we consider this pair as the potential vulnerable-patched function pair and put them in the candidate set.

B. Vulnerability Search Combined With Patch Information

In this search stage, we use the machine learning algorithm as our classifier. We will introduce the digital feature extractor first. Each function holds a lot of digital information or metadata, such as the number of instructions and so on. The digital feature extractor is designed to extract a set of digital features, which can represent the binary function. The paper[10] indicates the number of instructions, the size of local variables, the number of parameters, the number of CFG-based blocks and the numbers of edges can be extracted as the digital features. We also classify and record the instructions according to the instruction types, like arithmetic instructions, data transfer instructions, logic operations instructions, function call instructions, function jump instructions and so on. Different from the paper[10], we calculate the SimHash of each function which is added to the digital features. Then we use the machine

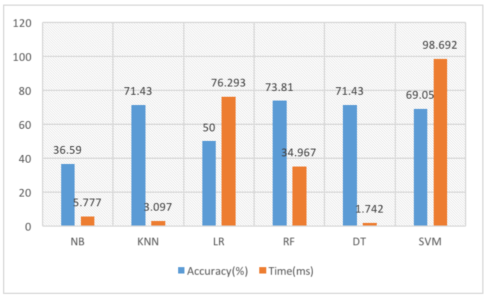


Fig. 2: The Performance of Machine Learning Algorithm

learning algorithms to classify the functions into different types according to the vulnerable-function database and the patched-function database. Considered the performance of each machine learning algorithm(fig2), we finally use the KNN algorithm as our classifier and we modify it to adapt our system. We check the results returned from the classifier. If each result has the similarity over 90% with the test function, then we still put them into the result set, or drop them. In this case, we can adapt to the real situation that not every unmatched-function pair with the similarity over 80% is the vulnerable-patched function pair.

We can also explain the rationality of the low false rate by double validation. Though the digital feature can’t represent a specific function, the results returned from the machine learning classifiers include the true function type. If the true function type is in each result, the intersection must have the true label. Thus, we can reduce the range of the candidate through making an intersection of two results returned from two classifiers respectively.

IV. Experiment

Our experiment is carried out in the system Ubuntu 14.04, running in an angr virtual environment[11][12][13] . The language we use is Python. We firstly introduce the two databases’ construction. Then, we compare some machine learning algorithms performance to get the best one. Then, we modify it to adapt our system. At last, we introduce some true examples which have the CWE vulnerabilities to verify the accuracy and compare performances between double validation and single validation.

A. The Construction of Databases

We have downloaded the common vulnerable functions on CVE website, which contains the corresponding patched function, including Stack Based Buffer Overflow, Heap Based Buffer Overflow, Buffer Underweite, Integer Overflow, Use after free, Double Free, Type Confusion and other vulnerabilities, about totally 148 kinds, more than 4000 vulnerable-functions and more than 5000 patched-functions. After we get the source code, we have to compile them in different platform, here we just consider the Linux and Windows.

Because the instruction confusions have negative influence on the performance, we have to preprocess the data information. According to the feature we extracted and the Simhash, we just focus on instruction rearrangement and register renaming.

B. An Example with CWE843

At first, we take the program with CWE843 and its patched-file as the input respectively in the KNN classifier. Then we can get a list of results which are function-vulnerability pairs. As TABLE I and TABLE II shown:

Function	Type	T/F
plt._libc_start_main	Missing_Handle,· · · ,Infinite_Loop	F
plt.printf	Infinite_Loop,· · · , Reachable_Assertion	F
sub_40124e	Type_Confusion,Null_Deref_From_Return	T
sub_400f80	Type_Confusion,Null_Deref_From_Return	T
sub_4012a0	Type_Confusion	T
sub_400f2e	Type_Confusion	T
. . .	. . .	. . .
_init	Missing_Handle,· · · , Reachable_Assertion	F
_libc_csu_init	Infinite_Loop,· · · , Reachable_Assertion	F

TABLE I: The Final Result of unpatched-program

T represent the true type is in the Type set; F represent the true is not in the Type set.

Function	Type Set	T/F
_init	Mismatched_Memory_Management	F
_libc_csu_init	Infinite_Loop,· · · , Reachable_Assertion	F
sub_40101c	Type_Confusion	T
sub_4010e2	Type_Confusion	T
sub_401175	Type_Confusion,Reachable_Assertion	T
sub_400e57	Type_Confusion	T
· · ·	· · ·	· · ·
printLine	Missing_Handle,· · · , Reachable_Assertion	F
deregister_clones	Uncontrolled_Recursion	F

TABLE II: The Final Result of patched-program

T represent the true type is in the Type set; F represent the true is not in the Type set.

The vulnerable program has 51 functions totally, and the classifier returns 16 functions that may be the potential vulnerability function. However only 4 functions have bugs. Also, there are 51 functions in the patch file with 4 bug functions, however 18 functions are returned from the classifier. Through observing the data, we find that there are nine functions are same which are _init, _libc_csu_init,· · · , printLine. They cost us too much energy to analyse, however they can't be the vulnerable functions.

Then, we put the two test programs into the SemHunt. We get a list of functions which have the similarity over 80% from the unmatched set. Thus we get the candidates that may be the vulnerable-function and patched-function pair as TABLE III shown. Then, we use these candidate pairs as the test function pair, and put them into the KNN classifier to get the potential vulnerability types set. At this time, there are only eight pairs left including the four vulnerable function pairs: sub_40124e:sub_40101c, sub_400f80:sub_4010e2, sub_4012a0:sub_401175, sub_400f2e:sub_400e57. Then through the intersection of each pair, we get a more accurate result.

U.Fun	P.Fun	C.Type	T.Type
sub_40124e	sub_40101c	Type_Confusion	Type_Confusion
sub_400f80	sub_4010e2	Type_Confusion	Type_Confusion
sub_4012a0	sub_401175	Type_Confusion	Type_Confusion
sub_400f2e	sub_400e57	Type_Confusion	Type_Confusion
· · ·	· · ·	· · ·	· · ·

TABLE III: The Final Result of CWE843

[1] U.Fun: Unpatch function, P.Fun: Patch function, C.Type: Candidate Type, T.Type: True Type

V. Conclusion And Future Work

We presented a system to efficiently identify already known vulnerability with the patch file information in binary code across two operating systems. In the preparation phase, two code bases of known vulnerability functions and corresponding patched functions are analyzed and their numeric features and SimHash are stored. When a new vulnerability and its corresponding patch are published, we always can't know the detail information about them. Our approach employs a three-stage method to quickly identify the vulnerability type of the program and which function is the buggy function. The first stage relies on the binary comparison technique to get the function-pairs which may be the vulnerability function and corresponding patch function. Then at the second stage, we extract the digital features and SimHash to retrieve very similar functions based on the KNN algorithm. These functions serve as candidates to the next stage. In the end, we make an intersection of the two candidates set to get the potential vulnerability types. Overall, our method can reduce the false positive and the cost of time doesn't increase too much.And we implemented our methods in a tool call SemHunt and evaluated its efficacy on several program and their patch files with CWE vulnerabilities.

Acknowledgment

We would like to thank Yong Tang for detailed discussions and suggestions about the algorithms used in this paper, and the reviewers from our laboratory for providing useful comments on this paper.

References

[1] J. Ming, M. Pan, and D. Gao, *iBinHunt: Binary Hunting with Interprocedural Control Flow*. Springer Berlin Heidelberg, 2012.

[2] Z. Wang, K. Pierce, and S. Mcfarling, "Bmat – a binary matching tool for stale profile propagation," vol. 2, p. 2000, 2002.

[3] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security, International Conference, ICICS 2008, Birmingham, Uk, October 20-22, 2008, Proceedings*, pp. 238–255, 2008.

[4] K. Riesen, M. Neuhaus, and H. Bunke, *Bipartite Graph Matching for Computing the Edit Distance of Graphs*. Springer Berlin Heidelberg, 2007.

[5] H. Flake, "Structural comparison of executable objects," in *IEEE Conference on Detection of Intrusions and Malware Vulnerability Assessment*, pp. 161–173, 2004.

[6] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," 2005.

[7] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," 2013.

[8] L. Liu, B. S. Wang, Y. U. Bo, and Q. X. Zhong, "Automatic malware classification and new malware detection using machine learning*,"

[9] J. Feist, L. Mounier, and M. L. Potet, "Statically detecting use after free on binary code," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.

[10] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *The Network and Distributed System Security Symposium*, 2016.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[12] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.

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