Computer Security Symposium 2019
21 - 24 October 2019

Similarity Based Binary Vulnerability Detection

Zeming Tai[1] Hironori Washizaki[1] Yoshiaki Fukazawa[1] Yurie Fujimatsu[2]
Jun Kanai[2] Yoshikazu Hanatani[2]

Abstract: This paper describes a scalable method of vulnerability detection for binary code based on the similarity between binaries. Every binary procedure is decomposed to several comparable strands, and those strands will be transformed into a normalized form by the optimizer, which allows determining similarity between two binary procedures through hash value comparison. The similarity based on hash value makes it possible to find semantic equivaluence in strands without using any debug information. Due to the low computational complexity of hash value comparison, analyzing binaries compiled from million lines of source code is feasible within an acceptable time. The high accuracy of vulnerability detection is achieved by using IR re-optimization and normalization method to eliminate the binary difference introduced by compile configurations. We have implemented our method and applied it to detect several existed vulnerabilities, including Heartbleed and Shellshock.

Keywords: binary analysis, binary code search, binary similarity, static analysis

1. Introduction

For a security researcher, it always takes plenty of time to identify a procedure from a binary since a binary can dramatically change when it is compiled using a different configuration like a different compiler and optimization level. Every small change in setting will bring vast differences into the assembly code. Generally, those differences don't affect a program's behavior but impede the process that researchers analyze a binary.

For some situations like locating a newly discovered zero-day threat, it can be susceptible to the time researchers spend on binary analysis because one more hour late may cause substantial economic losses. For example, the Heartbleed bug disclosed in 2014 is a severe vulnerability in a widely used implementation of the Transport Layer Protocol named OpenSSL. In the case, some traditional time-consuming binary analysis methods may not suitable for fast address the vulnerable code in binaries.

We propose a scalable method of vulnerability detection for binary code named Razor-V. This method is based on the similarity between binary procedures through hash value comparison and was first proposed in the paper [1]. We made some modifications to this technique to allow it works on pure VEX-IR and combine several metrics to enhance the detection result. The instrument consists of three fundamental parts.

The first part is re-optimizing the assembly code to the highest optimization level, which aims to unify the binaries' optimization levels. The second part is responsible for program slicing. Every procedure's assembly code will be sliced to several comparable strands according to the data dependence. The third part processes the strands to a standardized form by normalization and canonicalization. Furthermore, to get a more precise detection result with a low false-positive rate, we apply the TFIDF method on hash value to assign every strand with a suitable weight that can correctly reflect every strand's rarity.

We address the following research questions.

RQ1 Is VEX-IR a suitable language for re-optimization based vulnerability detection?

RQ2 Can mismatch be solved without introducing another tool?

RQ3 Can Razor-V provide a high-speed detection without loss of accuracy?

2. Background and Problem

In this section, we note several existed approaches to detect vulnerabilities and introduce three main types of problems we have met in research.

Generally, the existed sliced-based detect approaches can be classified by the way they define the similarity between two procedures. Some of them evaluate similarity by using a program solver to compare two procedures' behavior. The problem with this method is the time-efficiency. Detecting vulnerability in a real binary usually requires the tool to search thousands of procedures, which a program solver may take tens of hours to finish. Others use an optimizer to transform slices into a uniformed representation and directly compare their hash values. However, it is hard to select a proper kind of IR (Intermediate Representation) to achieve that. LLVM-IR is the most widely used, but it does not have an official lifter, which means some 3rd party lifters have to be introduced, which cannot ensure the correctness of the lifted result. VEX-IR has its official lifter, but it does not have a powerful optimizer compared with LLVM-IR. A compromise is lifting the binary to VEX-IR, then translate it to LLVM-IR, which takes time and there is not any widely used translator proved to be reliable.

The main goal of this paper is to eliminate the differences that are introduced by compile configurations using pure VEX-IR.

1 Waseda University.

2 Toshiba Corporation.

© 2019 Information Processing Society of Japan

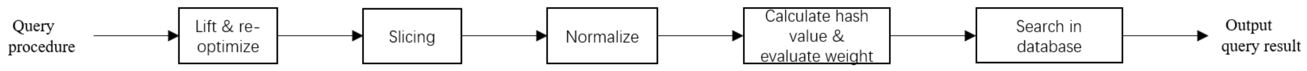


Figure 1. An overview of our vulnerability detection chain. Taking a query procedure as an input, the similarity between procedures can be evaluated through a lift, re-optimization, normalization, and rarity estimation. Once the features of the query procedure have been calculated, it will be compared with all procedures' features to evaluate the matching score and match proportion for each procedure.

As shown in Figure 2, the assembly codes compiled from the same source code become significantly different with each other in register address and instruction selection, but keep semantically consistent. The forms of those differences may be vast, but generally, they can be classified into three types.

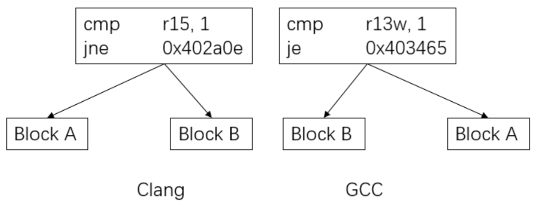


Figure 2. Differences brought by compile configurations

Semantically equivalent code: It is mainly caused by different instruction selection. For example, when a compiler tries to compile a conditional jump, it has to use some instructions corresponding to compare functionality to check if its condition is satisfied. For the most widely-used instruction set, they all provide many alternative compare instructions like cmp and cmn for direct compare and compare negative. Each compiler with a different setting has its instruction selection preference, then binary difference generates.

Arbitrary register use: For different compilers, vast register allocation strategies will be applied in various scenarios. Different architectures also affect the registers that compilers use; each of them has a unique register naming method and hardware resources. For example, x86-64 name its register by the r follows a number, but ARM® prefers x follows a number.

Optimization level variance: Optimization level variance can be found in different scenarios. Binaries from a development environment usually have a low optimization level, which is more convenient to understand the relationship between assembly code and programming language. Released binaries are usually highly optimized to get an efficient performance. Generally, binaries are compiled with a suitable optimization level to fit designers' requirements, which will dramatically affect the generated assembly code.

3. Proposed approach

We proposed a scalable method for detecting procedures that are compiled from the same source code in binaries. Moreover, by combining several different metrics, we correct mismatched cases that are due to insufficient re-optimization. Figure 1 gives an overview of our vulnerability detection chain. Given a procedure from a binary as a query procedure and a set of binaries as search areas, we aim to detect all procedures in binaries that are close to the query procedure by evaluating the similarity between them. Several essential steps are described as follows：

Step1, lifting assembly code into VEX-IR & Re-optimization. We use the existed tool name Angr to lift binaries into VEX-IR. The main reason for using VEX-IR is that it has an official lifter, which can ensure the correctness of the lifting result and Angr provides full functional toolchains to analyze VEX-IR. Moreover, VEX-IR has been widely used in some industrial level products like Valgrind, which proves its dependability.

Step2, decomposing procedure into comparable strands. Directly using the whole query procedure to match always requires some heavyweight program verifier and is very easy to fail due to the fact that many compile configurations can differ the output binaries. Then to achieve a fast speed detection method, the only viable method is to decompose the query procedure to several comparable units, each has isolated semantic information. We use the slicing method based on data dependency, which is fully discussed in paper [4]. For each strand, it gives every variable a complete data dependency with a backward direction until strands can cover all the code of the procedure.

Step3, normalizing strands into a standardized form. Strands cannot be directly used for matching because they still contain several types of differences that are introduced by compile configuration. There are several steps of normalization that have to be applied onto strands, including variable name, address, offset, function name normalizations. After all these steps, strands are qualified to match.

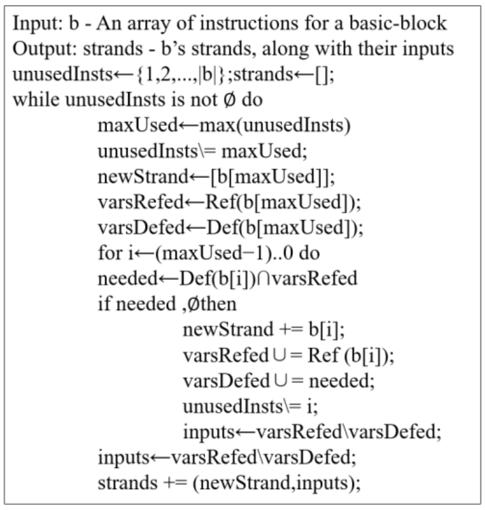
Step4, evaluating similarity score. Two procedures' matching score can be expressed as the sum of all matched strands. However, giving every strand with the same weight can result in a false detection result due to that common strands contribute too much score. A suitable solution for this is assigning every strand with a weight to limit the effect brought by common strands. The weight can be defined by TFIDF (Term Frequency-Inverse Document Frequency), which reflects the rarity of the strand. After applying the weighted score method, strands can contribute different scores corresponding to its rarity.

For VEX-IR, in some cases, due to insufficient re-optimization, it is hard to transform semantically equivalent strands into a syntactically equivalent form. In this case, many procedures are not completely matched, which means those procedures contains rare strand by coincidence may mismatch even if there is a huge gap between the size of the two procedures. Taking matched proportion into account can effectively decrease mismatch cases due to poor re-optimization.

4. Implementation details

4.1 Program Slicing

The slicing method used in this paper is first proposed in this paper [4]. This method aims to construct several unrelated strands each give one variable a complete data dependence. Cause the method's decomposing sequence is from the end of the code to the beginning of the code, it is named as backward slicing. There are some other slicing algorithms like forward slicing, backward slicing with partial data dependence, and isomorphic subgraph, which is described in the paper [5]. For decomposing a procedure into several comparable units, backward slicing performs well and avoid generating too many tiny strands. The detail of backward slicing is shown in Figure 3.

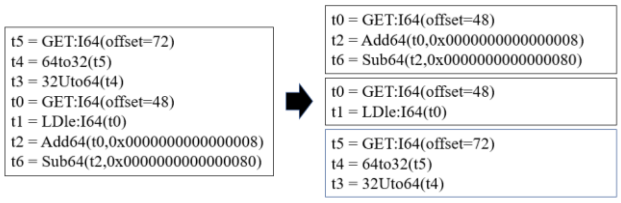


```
Input: b - An array of instructions for a basic-block
Output: strands - b's strands, along with their inputs
unusedInsts←{1,2,...,|b|};strands←[];
while unusedInsts is not Ø do
        maxUsed←max(unusedInsts)
        unusedInsts\= maxUsed;
        newStrand←[b[maxUsed]];
        varsRefed←Ref(b[maxUsed]);
        varsDefed←Def(b[maxUsed]);
        for i←(maxUsed−1)..0 do
        needed←Def(b[i])∩varsRefed
        if needed ,Øthen
                newStrand += b[i];
                varsRefed∪= Ref (b[i]);
                varsDefed∪= needed;
                unusedInsts\= i;
                inputs←varsRefed\varsDefed;
        inputs←varsRefed\varsDefed;
        strands += (newStrand,inputs);
```

Figure 3. The implementation of backward slicing.

The algorithm starts to slicing the program at the end of code and iterate all instructions backward. The instruction referred variables and defined variables can determine whether the instruction is included in a new strand or not. When the algorithm generates one strand for one loop, and it keeps iterating all instructions until there are none unused instructions in the procedure. Figure 4 shows an example of backward slicing.

The drawback of backward slicing is that the size of strands is usually larger than the size of the original procedure due to that occasionally some instruction in different strands refers to the same previous instruction, then two strands contain duplicated code. This case can be solved by making a small modification on the original algorithm to iterate not all instructions but unused instructions in the procedure, then the size of strands is strictly equal to the size of the original procedure. However, after that, not all the strands contain complete data dependence, some of it only have partial data dependence to remove duplicated code.



```
t5 = GET:I64(offset=72)
t4 = 64to32(t5)
t3 = 32Uto64(t4)
t0 = GET:I64(offset=48)
t1 = LDle:I64(t0)
t2 = Add64(t0,0x0000000000000008)
t6 = Sub64(t2,0x0000000000000080)
```

```
t0 = GET:I64(offset=48)
t2 = Add64(t0,0x0000000000000008)
t6 = Sub64(t2,0x0000000000000080)
```

```
t0 = GET:I64(offset=48)
t1 = LDle:I64(t0)
```

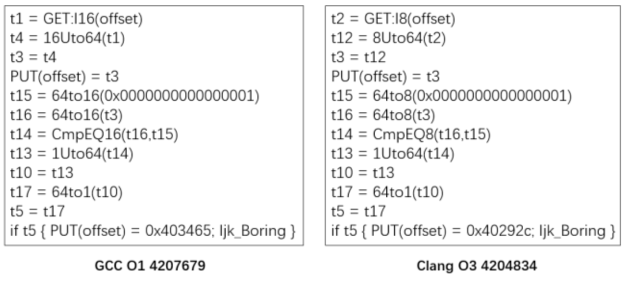
```
t5 = GET:I64(offset=72)
t4 = 64to32(t5)
t3 = 32Uto64(t4)
```

Figure 4. An example of backward slicing.

4.2 Strand Normalization

Strands normalization aims to eliminate the difference in variable naming, address, register, and datatype selection. Different compilers and compile configurations can result in different compile strategies, which can be reflected in the VEX-IR. Figure 4 gives an example of the compile difference.

The strands shown in Figure 4 are compiled from the same source code but with different compilers and optimization levels. Those differences result in different variable names, addresses, and data types. The semantical equivalence between two strands indicates that it is possible to normalize them to a uniformed representation.



```
t1 = GET:I16(offset)
t4 = 16Uto64(t1)
t3 = t4
PUT(offset) = t3
t15 = 64to16(0x0000000000000001)
t16 = 64to16(t3)
t14 = CmpEQ16(t16,t15)
t13 = 1Uto64(t14)
t10 = t13
t17 = 64to1(t10)
t5 = t17
if t5 { PUT(offset) = 0x403465; Ijk_Boring }
```

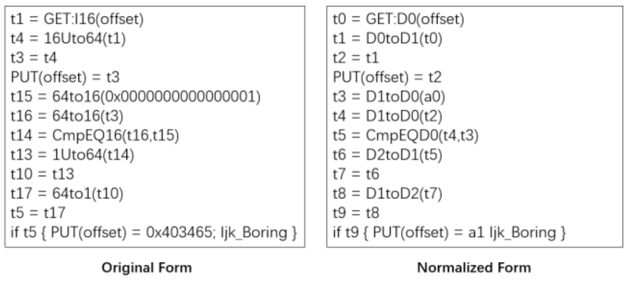
GCC O1 4207679

```
t2 = GET:I8(offset)
t12 = 8Uto64(t2)
t3 = t12
PUT(offset) = t3
t15 = 64to8(0x0000000000000001)
t16 = 64to8(t3)
t14 = CmpEQ8(t16,t15)
t13 = 1Uto64(t14)
t10 = t13
t17 = 64to1(t10)
t5 = t17
if t5 { PUT(offset) = 0x40292c; Ijk_Boring }
```

Clang O3 4204834

Figure 5. VEX-IR from different configurations

By renaming variable names, addresses, and data types by the sequence they appear in the code, a strand can be transformed into a normalized form, which eliminates all the difference introduced by compilers and configurations. A comparison between the original VEX-IR strand and the normalized strand is shown in Figure 5.



```
t1 = GET:I16(offset)
t4 = 16Uto64(t1)
t3 = t4
PUT(offset) = t3
t15 = 64to16(0x0000000000000001)
t16 = 64to16(t3)
t14 = CmpEQ16(t16,t15)
t13 = 1Uto64(t14)
t10 = t13
t17 = 64to1(t10)
t5 = t17
if t5 { PUT(offset) = 0x403465; Ijk_Boring }
```

```
t0 = GET:D0(offset)
t1 = D0toD1(t0)
t2 = t1
PUT(offset) = t2
t3 = D1toD0(a0)
t4 = D1toD0(t2)
t5 = CmpEQD0(t4,t3)
t6 = D2toD1(t5)
t7 = t6
t8 = D1toD2(t7)
t9 = t8
if t9 { PUT(offset) = a1 Ijk_Boring }
```

Figure 6. Strand Normalization

4.3 Similarity Evaluation

Benefit from the previous steps performed to re-optimize and normalize the strands, we can implement an efficient comparison between strands based on hash values. We can use a set of hashed strands to represent a given procedure p:

$$R(p) = \{MD5(Normaize(s_p)) | s_p \in p\}$$

The initial definition of similarity between the query procedure p and the target procedure q is that the similarity equals the intersection of *R(p)* and *R(q)*. However, the problem of this method is that common strands, and rare strands contribute the same to the similarity score, which makes it possible that some unrelated procedures can mismatch with the query procedure because of the same common strands they have. Therefore, it is necessary to introduce a new method to evaluate the significance of a strand. A common solution is TFIDF which provide a fair way to estimate the probability that a term appears in a document. Follows this idea, we can give our definition of similarity and significance:

$$S(p,q) = \sum_{s \in R(p) \cap R(q)} \frac{1}{Pr(s)} \tag{1}$$

where:

$$Pr(s) = \frac{f(s)}{|P|} \tag{2}$$

$$\mathrm{f(s)} = \begin{cases} |\{p \in P | s \in R(p)\}|, & \mathrm{s} \in \mathrm{P} \\ 1, & \mathrm{else} \end{cases} \tag{3}$$

We determine the significance of each strand by inverse probability in equation (1). Then as defined in equation (2), the probability of a strand can be estimated by its frequency *f(s)* divided by the number of all target procedures *P*. To get a more accurate estimate result, it is better to estimate the significance using a set of randomly selected procedures. Then the equation (1) and (2) can be updated to (4) and (5), which gives the equations a specific estimate field *W*.

$$S_W(p,q) = \sum_{s \in R(p) \cap R(q)} \frac{1}{Pr_W(s)} \tag{4}$$

$$Pr_W(s) = \frac{f(s)}{|P|} \tag{5}$$

According to the paper [1], setting the size of *W* to 1K is enough to get an accurate estimation of *Pr(s)*.

Using the similarity score discussed above as the only metric for vulnerability detection can achieve an accurate and robust performance when most of the strands can correctly match with the target procedure. However, for VEX-IR, there are not many usable re-optimize tools. A general reason for the failure of a vulnerability detection is the insufficient re-optimization that makes the strands cannot be correctly matched even after normalization. For those cases, one procedure that contains a rare strand may get a comparable high similarity score by accident, then a false positive happens. To solve this problem, we are going to introduce a new metric that can be used to avoid false positives by filtering the match result. Generally, although, a false positive case can get a comparably high score, its size can be very different from the query procedure. Therefore, a metric that can reflect the matched proportion can help us distinguish false-positive cases. We define the matched proportion in an intuitive form as follows:

$$P(p,q) = \frac{|R(p) \cap R(q)|}{|R(p) \cup R(q)|} \tag{6}$$

The similarity score combined with the matched proportion can give an accurate detecting performance even in a poorly matched environment with a slightly increased computational cost.

5. Evaluation

In this section, we evaluate our methods' performance, including how matched proportion improves the detection accuracy in a poorly matched environment, how our methods perform when detects a real vulnerability and a comparison with the program solver based tool ESH.

5.1 Hybrid Metrics

The metric similarity score can be reliable when the vulnerable procedure can match most of its strands. However, in some cases, the poor performance of re-optimizer makes it possible to generate several false positive.

Table 1. A comparison of similarity score and filtered similarity score in a detect experiment over 6000 procedures

Vulnerability	Similarity Score		Filtered Similarity Score	
	#FPs	#PFPs	#FPs	#PFPs
Heartbleed	0	2	0	0
Shellshock	2	12	0	0

As shown in Table 1, #FPs are for false positives, and #PFPs are for potential false positives, which those cases may be false-positive depending on the threshold for distinguishing vulnerable procedures against normal procedures. After using the matched proportion to filter the target procedures, we focus on finding vulnerable procedures in those procedures that get a high matched proportion by similarity scores. Benefit from the filtration, we exclude the effects of procedures contains rare strands on coincidence. We can draw this conclusion due to the fact that both of the FPs and PFPs decrease when the proportion filter method is applied to target procedures. It is acceptable to get a more accurate performance with slightly increased computational cost. This show that pure VEX-IR without introducing other language is enough for vulnerability detection (RQ1). Moreover, with taking an additional metric into account, the mismatched cases are significant decreased (RQ2).

5.2 Vulnerability Detection

Finding Heartbleed: We are going to use our methods to detect the vulnerability Heartbleed. The query procedure is the vulnerable procedure named tls1_process_heartbeat from the binary OpenSSL -1.01. The detail of this experiment is shown in Table 2.

Table 2. The configuration for #1 experiment.

	Name	Compiler
Query Procedure	tls1_process_heartbeat	Clang -O1
Detect Area	OpenSSL	GCC -O3

The detect area is the binary OpenSSL compiled by GCC with a different optimization level. For this experiment, we expect our method can detect two vulnerable functions for the binary, one is tls1_process_heartbeat and dtls1_process_heartbeat from the binary. Both of them are the cause of Heartbleed and share most of the code. Therefore, our method should detect both of them with only tls1_process_heartbeat as a query procedure. The result of detection shown as follows:

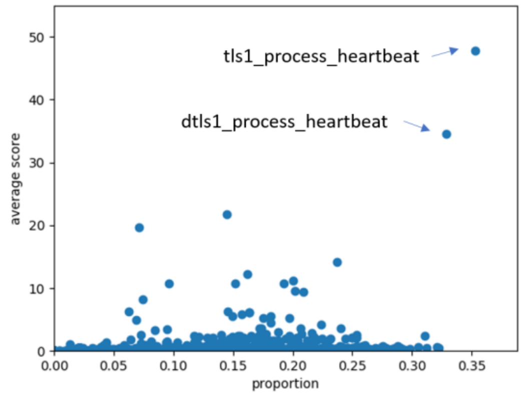


Figure 7. Result for Heartbleed

Due to our filter rules, we only find the vulnerable procedures at the rightmost side of the proportion axis and can easily find out two outliers corresponding to two vulnerable functions.

Finding Shellshock: The query procedure for Shellshock is parse_and_execute. We are going to detect the vulnerable function from the binary Bash which contains over 2000 functions. The configuration of this experiment is shown in Table 3.

Table 3. The configuration for #2 experiment.

	Name	Compiler
Query Procedure	parse_and_execute	Clang -O1
Detect Area	Bash	GCC -O3

The result of the detection is shown as follows:

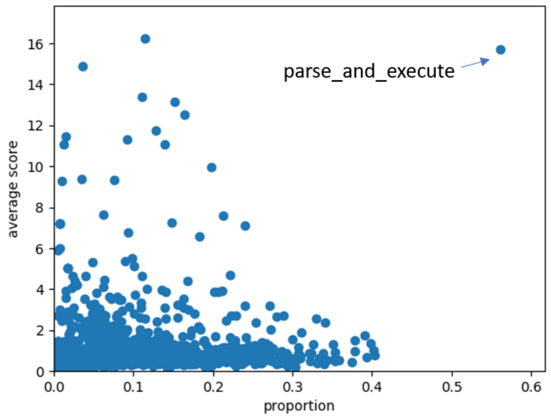


Figure 8. Result for Shellshock

Also, we can easily find the vulnerable procedure at the top of the rightmost side of Figure 7.

5.3 Performance Comparison

The main advantage of the search method based on comparisons of hash values is that the search efficiency is significantly higher than those search methods based on comparison using some program solvers. In this section, we will show some comparison between Razor-V and the ESH, a search tool which also takes strands as basic search units, but does a comparison using a program solver. We use the same test dataset with ESH which includes OpenSSL, Bash and so on. The detail of the data set can be found in the paper [2] and the home page pf ESH. All experiments have been performed on a machine with one Intel® i7-4870HQ (2.5GHz) processor (4 cores) running Windows® 10.

To keep the consistency of experimental with paper [1], we randomly pick up 1000 procedures from our dataset and use their strands to draw estimation of a strand's rarity. According to paper [1], 1000 procedures are enough amount to give every strand a proper weight when evaluating similarity. In Table [4], we show two experiments where two real vulnerability are used as query functions and detect against 1500 procedures.

Table 4. The comparison between Razor-V and the ESH.

CVE	Alias	Razor-V		ESH	
		ACC	Time	ACC	Time
2014-0160	Heartbleed	1	46s	1	19h
2014-6271	Shellshock	1	43s	0.998	15h

Since our hardware cannot afford a heavyweight task like ESH, each of the ESH performance tests takes more than two days to finish. In Table 4, we directly reference the accuracy and running time from paper [1] to avoid the suspicion that we degrade the ESH's performance. From the results of experiments, it is hard to conclude that re-optimization based hash value comparison is more accurate than the comparison based on a program solver. However, one thing is clear that our approach is much faster than the ESH without any accuracy losses, benefit from the fast speed matching (RQ3). The main reason for that is although nowadays program solvers are quite efficient, they still not suitable for some task that requires fast speed. On our hardware, match each procedure pair may take time varies from several minutes to several tens of minutes depending on the size of the procedure. Prefiltering the impossible pair by the difference of size can slightly improve the performance. However, the search speed is still slow compared with matching hash values.

6. Related work

This paper mainly concentrates on code re-optimization, slicing, and normalization. We have already mentioned some of the related works like the pioneer of using re-optimization to eliminate the difference caused by compile configuration [1]. [1] proposed a method of vulnerability detection using both VEX-IR and LLVM-IR. The approach described in [1] first lift binary to VEX-IR and then translate it to LLVM-IR to avoid the matching performance being limited by the lack of VEX-IR re-optimizer.

Compared with merely slicing a program by the data dependence, there are several advanced slicing methods, including tracelets and isomorphic subgraphs. The concept of tracelets can be described as continuous, partial traces of an execution path. In paper [3], it fully discussed how to use tracelets to build a similarity between procedures which is a more robust metric when compiles optimize the control flow of a program. This method combined both data dependency and control flow to improve slicing accuracy. Another method is using isomorphic graphs which can be regarded as an improved version of the traditional slicing method based on data dependence. In paper [5], data dependence graphs are constructed in which maximal isomorphic subgraphs will be detected to use as procedures matching proof. Several different methods to detect isomorphic subgraphs have been discussed in paper [5].

7. Conclusion and Future work

We proposed an approach to detecting vulnerable code in executable. This approach can be applied onto cross compilers, optimization levels scenario. We decompose procedures to comparable strands and achieve a fast speed matching using a hash function. To avoid false positives due to insufficient re-optimization, we introduce to a new metric to filter those strands that have a significant difference in size. Furthermore, we compare our approach with the ESH, which showing that compare with using program solvers to match strands, using hash values can provide a much faster performance without any accuracy losses.

VEX-IR has its official lifter, which ensures the correctness of the lifted result, but it doesn't have a powerful re-optimizer that supports out-of-context re-optimization. This requires some supplemental methods like adding more metrics to deal with insufficient re-optimization. Meanwhile, LLVM-IR as another backend language with multiple optimization tools is expected to work better than VEX-IR in re-optimization. As our future work, we plan to perform several experiences to compare VEX-IR and LLVM-IR in all aspects of procedures matching.

References

[1] David Y, Partush N, Yahav E, et al. Similarity of binaries through re-optimization[J]. programming language design and implementation, 2017, 52(6): 79-94.

[2] David Y, Partush N, Yahav E, et al. Statistical similarity of binaries[J]. programming language design and implementation, 2016, 51(6): 266-280.

[3] David Y, Yahav E. Tracelet-based code search in executables[J]. programming language design and implementation, 2014, 49(6): 349-360.

[4] Weiser M. Program Slicing[J]. IEEE Transactions on Software Engineering, 1984, 10(4): 352-357.

[5] Avetisyan A, Kurmangaleev S, Sargsyan S, et al. LLVM-based code clone detection framework[C]// Computer Science & Information Technologies. IEEE Computer Society, 2015.

[6] Sargsyan S, Kurmangaleev S, Belevantsev A, et al. Scalable and accurate detection of code clones[J]. Programming & Computer Software, 2016, 42(1):27-33.

[7] Yan S, Wang R, Salls C, et al. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis[C]// Security & Privacy. 2016.

[8] Ball T, Larus J R. Efficient path profiling[J]. international symposium on microarchitecture, 1996: 46-57.

[9] Bellon S , Koschke R , Antoniol G , et al. Comparison and Evaluation of Clone Detection Tools[J]. IEEE Transactions on Software Engineering, 2007, 33(9):577-591.

[10] Comparetti P M, Salvaneschi G, Kirda E, et al. Identifying Dormant Functionality in Malware Programs[J]. ieee symposium on security and privacy, 2010: 61-76.

[11] Khoo W M, Mycroft A, Anderson R J, et al. Rendezvous: A search engine for binary code[C]. mining software repositories, 2013: 329-338.

[12] Rosenblum N E, Miller B P, Zhu X, et al. Extracting compiler provenance from program binaries[C]. workshop on program analysis for software tools and engineering, 2010: 21-28.

[13] Bruschi D, Martignoni L, Monga M, et al. Detecting self-mutating malware using control-flow graph matching[J]. international conference on detection of intrusions and malware and vulnerability assessment, 2006: 129-143.

[14] Baker B S. On finding duplication and near-duplication in large software systems[C]. working conference on reverse engineering, 1995: 86-95.

[15] Ducasse S, Rieger M, Demeyer S, et al. A language independent approach for detecting duplicated code[C]. international conference on software maintenance, 1999: 109-118.

[16] Kaur R , Singh S . Clone detection in software source code using operational similarity of statements[J]. ACM SIGSOFT Software Engineering Notes, 2014, 39(3):1-5.

[17] Komondoor R, Horwitz S. Using Slicing to Identify Duplication in Source Code[C]// International Symposium on Static Analysis. 2001.