SemDiff: Finding Semtic Differences in Binary Programs based on Angr

Shi-Chao WANG[1,*], Chu-Lei LIU[2,a], Yao LI[3,b], and Wei-Yang XU[4,c]

[1]*National University of Defense Technology, China*
[2]*Haerbin Engineering University, China*
[3]*National University of Defense Technology, China*
[4]*National University of Defense Technology, China*
[a]*634390770@qq.com,* [b]*Faith_ly@yeah.net,* [c]*286406726@qq.com*
**Corresponding author: wshchao@163.com*

Abstract: We introduce SemDiff, a novel technology for finding semantic differences between two binary files. Now, the vendor will release the information to patch the previous version which has vulnerability. Then, we can compare the differences and similarities between the two versions to get the unpublished details of the 1day vulnerabilities. Tools, such as BinDiff, BinHunt and iBinHunt ,have worked on this project before, however , there are some weaknesses on them. Just like BinDiff, a comparison method based on structure, can not be effective for judging the semantic differences. Though the other two tools(BindHunt and iBinHunt) can recognize the differences we focus on, they can not effectively verify the functional inlining and spend a pretty long time to finish the process because the use of graph-based isomorphism algorithm. In the paper, we first propose SemDiff, which uses the existing tool(angr) to generate the intermediate language(VEX). Then, because of the nature of program, the data read from and written to the memories, we record these information to implement the comparison. Last, an improved BinDiff algorithm is used to match the basic blocks. In this paper, we take some real vulnerabilities as examples, such as CVE-2010-3974-Microsoft Windows to test our tool, reaching a good goal, matching more blocks than BinDiff and taking less time than BinHunt and iBinHunt.

1 Introduction

Now, for the purpose to protect the source code, many software vendors make the source code of their programs unavailable and when the vulnerabilities occurs, the patch is released in binary mode, rather than the source code. As Microsoft and other companies, when they publish a patch, no details are showed [1]. This situation increases the difficulty in analyzing the potential vulnerabilities to protect us from those threats, which may hijack our data, steal our privacy and so on. So, it is significant for us to understand the differences between the two versions. And, finding out the 1day vulnerabilities is one of the roles of the SemDiff.

However, because of instruction obfuscation, mutation optimization technology and other practical problems, the binary comparison is difficult. Modifying the software process call graph, using the Proxy point, changing the function symbols, sharing basic blocks, adding entry points, re-allocating registers, instruction sequence replacement, all increase the difficulties.

In addition, there are may prior works for solving these problems. Almost four types of methods have been developed for automatically comparing the structural similarities of executables, the class of BinDiff[2-4], Fingerprint and String Hashing[1,11,12], Bipartite Graph matching such as GED[1,13] and other graph methods[6,7]. The detailed introduction is showed as follows.

In this paper, we propose a new method called SemDiff to find the semantic differences between the two programs. Our method, compared with previous methods, such as BinDiff and BinHunt, provides a kind of rapid and accurate matching. For BinDiff, it only relies on structural information, which leads to many unmatched functions that should be matched. Our method is based on the control flow on basic blocks, symbolic execution[8] and the theorem prover. We first construct the intermediate representation of the program(we used the VEX there), then generate the control flow graph of the basic blocks. After that, we record the data written to and read from memories and registers, and then we put them into theorem prover to judge the similarity. Last, we use these information and the SemDiff to match the blocks.

Our approach is an interprocedural analysis rather than an intraprocedural analysis. Intraprocedural analysis limit to the scope of the current function, however, interprocedural analysis has the ability to enter sub-functions [9]. Needless to say, in-process analysis is much simpler than the process. However , sometimes we want to follow the path to find what we want, they may occur in different functions.

Our aritcles is organized as follows: Section 2 introduces the whole framework and an overview of each

part of the SemDiff, the next Section 3 shows the SemDiff algorithm which is improved from BinDiff, then we present the experiment and the analysis of them in Section 4. The limitation and future work are summarized in Section 5.

2 System Architecture

Fig.1 shows the whole system architecture. Firstly, the two binary files are loaded into angr platform, and angr runs its own disassembler, converting the execution files to assembly code. Then, they become the intermediate representation blocks through the IR convertor, which is then loaded into the CFG constructor to generate intermediate language representations control flow graph. After that, using the symbol execution to record the data which are judging by theorem proving. Last put the data into the SemDiff to get the matched set.

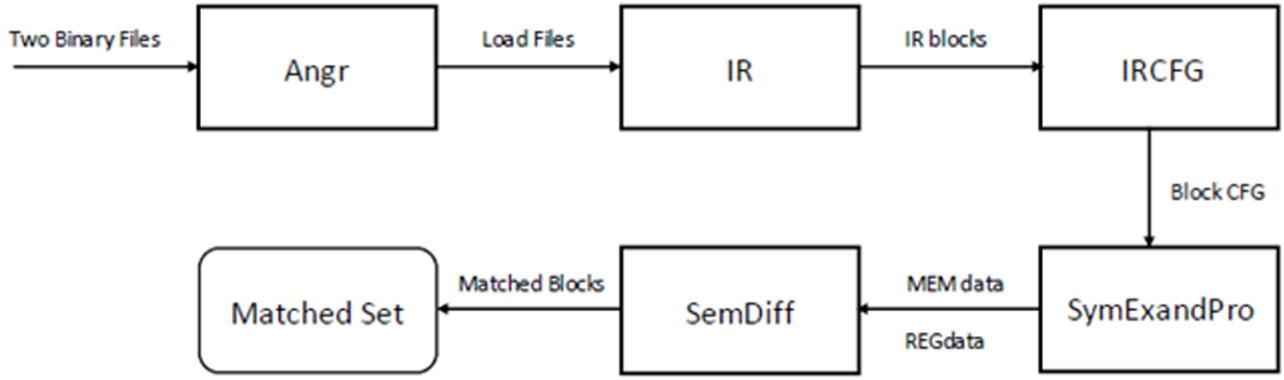

Figure 1. System architecture of the SemDiff

2.1 Angr

The writer of angr wants to create a user-friendly binary analysis suite, allowing a user to simply start up iPython and easily perform intensive binary analyses with a couple of commands. That being said, binary analysis is complex, which makes angr complex. The more details can be seen in [10].

2.2 Intermediate Representation

In VEX, the code is broken down into smaller blocks which is called IRSBs. IRSB is a single entry and multiple exit, which contains three elements:1.a type environment; 2.a list of statement; 3.a jump that exits form the end of the IRSB.

2.3 Symbolic Execution and Theorem Proving

Symbolic execution is a well known technique for representing scalar program analysis with symbolized values. We can perform symbolic execution to get the results of each step, and record the contents which are written to each memory and register. Because the core of executing the program is the data which are read from or written to the memory and register. So we use them to represent the semantic performance of a basic block. To determine the two basic blocks are functionally identical, we define a rule as follows:

definition1：（pair basic blocks equivalence formulas of data）Given two list of data that are recorded in the memories and registers: X = [$x_0, x_1, \ldots, x_m$],and Y = [$y_0, y_1, \ldots, y_n$]. For every x_i in X, there is a bijection, that, y_j = f(x_i) is existed ,then ,we call the data is same. The formula is described as follows:

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

3 SemDiff Algorithm

Now, we propose our SemDiff as three parts(WholeMatch, MatchPro, SemDiff).As with the previous algorithms, at the beginning of the initially matching process, we find the blocks which are uniquely matched, what is said that, we find some basic blocks that have the same symbol expressions in each addresses. Because there may some different blocks with the same data. This is also well explained in practice. In different functions, there are some similar blocks, for different functions may call the same part of the function.

Algorithm 1: Whole Match

```
1  function wholeMatch(S_A, S_B):
2  M ← ∅
3  foreach a_i ∈ S_A and b_j ∈ S_B do:
4      if ∃f(a_i) = b_j then:
5  M ← M ∪ (a_i, b_j)
6  S_A ← S_A \ a_i
7  S_B ← S_B \ b_j
8  break
9  return (M, S_A, S_B)
```

Figure 2. Algorithm 1

Algorithm 2: Match propagation

1 **function** MatchPro(M, S_A, S_B):
2 **foreach** $(a_i, b_j) \in M$ **do:**
3 $M' = M$
4 $SubS_A = FindPaAndCh(a_i)|$
5 $SubS_b = FindPaAndCh(b_j)$
6 $SubM', S_A, S_B = wholeMatch(SubS_A, SubS_B)$
7 $M' = M' \cup SubM'$
8 $M = M'$
9 **return** (M, S_A, S_B)

Figure 3. Algorithm 2

For the first part of the algorithm(Fig.2), S_A and S_B are the basic block sequences of the two programs partly. At the beginning, we assign the empty set to the Match set, because we do not perform the matching process. Then, the third line, we check if each a_i in Sa has the only match b_j in Sb, and whether a_i is the only match to b_j. If there is a bijection existing, add the pair (a_i, b_j) into the match set and delete the elements a_i, b_j from the S_A, S_B respectively.

Then, it is the second part (Fig.3). We call it the match propagation, because it is applied in the propagation progress. The input of the second part is the match set which is got from the first step and the remaining unmatched basic block sequence (S_A, S_B). Then we match each block from a small set, which is the line 4 and line 5 shown. $SubS_A$ is the set of the parent nodes and children nodes to the matched block a_i, also $SubS_B$ is the set of block b_j. After we get the subset, we use them into the wholeMatch to get the matched blocks. The reason why we use this way is the remaining blocks after the step one are the blocks with the different data or more than two blocks have the same symbol expressions. Then, we reduce the scope of the block lists to find more matching blocks through the context of the matched blocks.

Algorithm 3: SemDiff

1 **function** SemDiff$(Path_A, Path_B)$:
2 $S_A \leftarrow Path_A$
3 $S_B \leftarrow Path_B$
4 $M' \leftarrow \emptyset$
5 (M, S_A, S_B) = wholeMatch(S_A, S_B)
6 **while** $M' \neq M$:
7 $M' = M$
8 M, S_A, S_B = MatchPro(M, S_A, S_B)
9 $M, S_A, S_B = wholeMatch(S_A, S_B)$
10 **return** (M, S_A, S_B)

Figure 4. Algorithm 3

The last part is a loop (Fig.4), which is the whole process of our SemDiff algorithm. The main core of the SemDiff is the line 8 and line 9. After the line 8, we may get two remaining lists which may have the unique blocks with the same symbol expressions but they are not the parent node or the child node of the matched blocks. If we do not execute the line 9, we may miss some of them.

4 Experiment

Our experiment is carried out in the system Ubuntu 14.04, running in an angr virtual environment. The language which we use is Python. We first run a sample program. The sample's input is the same two paths. Our purpose is to illustrate our SemDiff's efficiency. Then we run two real program with vulnerability. The inputs are the paths of unpatched and patched program respectively.

4.1 The sample

We test a simple sample named fauxware, and finally find all the blocks matched with score 1 experienced 4 times during 45.301483 Seconds.

4.2 CVE-2010-3974

We first run the unpatch version and patch version, and change them into IR blocks, like Fig.5.

```
0x400774:   lea  rax, qword ptr [rbp - 0x20]
0x400778:   mov      edx, 8
0x40077d:   mov      rsi, rax
0x400780:   mov      edi, 0
...
0x400785:   call 0x400530
```

```
00 | ------ IMark(0x400774, 4, 0) ------
01 | t4 = GET:I64(bp)
02 | t3 = Add64(t4,0xfffffffffffffe0)
03 | PUT(rax) = t3
...
16 | t10 = Sub64(t8,0x0000000000000080)
PUT(rip) = 0x0000000000400530; Ijk_Call
```

Figure 5. Assembly instruction and IR instruction

Then, we preprocess the data by ignoring the address of memory and register like Fig.6.

```
<BV640x100000e>
<BV640#56(mem_8+(7*reg_10_64[7:0]))>
<BV64reg_38_64+0xfffffffffffffe0>
<BV64__add__(reg_38_64,0x8,0x8)>
```

Figure 6. Final Data

We then run the basic block comparison. We finally find some matched blocks with score between 0.8 and 0.9. We then check the content. According to the knowledge, we find the vulnerability point existing in these blocks. One of the blocks in patched version is illustrated as Fig.7.

```
lea   ecx,ss:[ebp+var_1C]
push ecx
push ss:[ebp+arg_0]
...
jz loc_100C1DD
```

Figure 7. Blocks with score 0.8-0.9

We then trace these blocks path and finally find out the vulnerability in the unpatch version.

Summary

In this paper, we introduce a novel technique named SemDiff. It is based on the angr platform. The main technique we use is the symbol execution, Theorem Proving and the updated the comparison algorithm. Though SemDiff has worked well on some files, it depends on the angr platform seriously. So sometimes it can not well support the PE file. And the way to find the vulnerability is still manual operations. We will develop our tools to find the vulnerability automatically in the future.

Acknowledgement

We would like to thank Thomas Dullien for his fascinating seminar on BinDiffthat inspired this study.

References

1. Martial Bourquin, Andy King, Edward Robbins,BinSlayer :Accurate Comparison of Binary Executables(2013).
2. T. Dullien and R. Rolles. Graph-based comparison of executable objects. In Symposium sur la Se´curite´ des Technologies de l'Information et des Communications (2005).
3. Halvar Flake. More fun with graphs. Black Hat Federal (2003).
4. Halvar Flake. Structural comparison of executable objects. In Proceedings of the IEEE Conference on Detection of Intrusions and Mal- ware & Vulnerability Assessment - DIMVA, 2004,pp 161–173.
5. S. Bardin, P. Herrmann, and F. Ve´drine. Refinement-Based CFG Reconstruction from Unstructured Programs. In VMCAI, volume 6538 of LNCS, 2011, pp 54–69.
6. D. Gao, M.K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In Poceedings of the 10th International Conference on Information and Communications Security (ICICS 2008), 2008.
7. Jiang Ming, Meng Pan, and Debin Gao ,iBinHunt: Binary Hunting with Inter-Procedural Control Flow,Springer Berlin Heidelberg, 2013, 7839:92-109.
8. King, J.: Symbolic execution and program testing. Communications of the ACM 19(7) (1976).
9. Josselin Feist, Laurent Mounier and Marie-Laure Potet,Statically detecting use after free on binary code, Journal of Computer Virology and Hacking Techniques (2014).
10. S Yan，C Kruegel，G Vigna，R Wang(State of) The Art of War: Offensive Techniques in Binary Analysis, IEEE Symposium on Security and Privacy,(2016).
11. J. Oh. Fight against 1-day exploits: Diffing Binaries vs Antidiffing Binaries. In Black Hat USA, 2009. http: //www.blackhat.com/presentations/bh-usa-09/OH/ BHUSA09-Oh-DiffingBinaries-PAPER.pdf.
12. J. Oh. ExploitSpotting: Locating Vulnerabilities Out Of Vendor Patches Automatically. In Black Hat USA, 2010. http://www. darungrim.org/Presentations.
13. K. Riesen, M. Neuhaus, and H. Bunke. Bipartite graph matching for computing the edit distance of graphs. In Graph-Based Representa- tions in Pattern Recognition, volume 4538 of Lecture Notes in Computer Science, pp 1–12. Springer, 2007.