BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection

He Huang
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, QC, Canada
hua_he@concordia.ca

Amr M. Youssef
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, QC, Canada
youssef@concordia.ca

Mourad Debbabi
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, QC, Canada
debbabi@concordia.ca

ABSTRACT

Code reuse detection is a key technique in reverse engineering. However, existing source code similarity comparison techniques are not applicable to binary code. Moreover, compilers have made this problem even more difficult due to the fact that different assembly code and control flow structures can be generated by the compilers even when implementing the same functionality. To address this problem, we present a fuzzy matching approach to compare two functions. We first obtain an initial mapping between basic blocks by leveraging the concept of longest common subsequence on the basic block level and execution path level. We then extend the achieved mapping using neighborhood exploration. To make our approach applicable to large data sets, we designed an effective filtering process using Minhashing. Based on the proposed approach, we implemented a tool named BinSequence and conducted extensive experiments with it. Our results show that given a large assembly code repository with millions of functions, BinSequence is efficient and can attain high quality similarity ranking of assembly functions with an accuracy of above 90%. We also present several practical use cases including patch analysis, malware analysis and bug search.

CCS Concepts

•Security and privacy → Software security engineering; •Social and professional topics → Software reverse engineering;

Keywords

binary code reuse; binary code similarity comparison; patch analysis; bug search; malware analysis;

ASIA CCS '17, April 02-06, 2017, Abu Dhabi, United Arab Emirates
© 2017 ACM. ISBN 978-1-4503-4944-4/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3052973.3052974

1. INTRODUCTION

Reverse engineering is a primary step towards understanding the functionality and behavior of a software when its source code is not available. However, reverse engineering is a tedious and time-consuming process, and its success depends heavily on the experience and knowledge of the reverse engineer. Moreover, as the software to be analyzed grows in size, this task becomes overwhelming. Code reuse detection is thus of a great interest to reverse engineers. For example, given a binary and a repository of already analyzed and commented code, one can speed up the analysis by applying code reuse detection to the binary to identify identical or similar code in the repository, and then focus only on the new functionality or components of the binary.

Consider, for instance, malware reverse engineering. Malware authors do not create viruses from scratch; instead, they tend to reuse their existing source code. Besides, in order to not reinvent the wheel, they may leverage some open source projects that provide certain functionality that they require. Identifying those reused code not only greatly reduces the efforts of analysis, but also helps in understanding the behavior of malware. For example, Citadel, which is allegedly derived from the leaked Zeus source code, keeps most of the core components of Zeus intact [20], and the malware Flame makes use of the light-weight database engine SQLite [10].

Code reuse detection is also of high interest to software maintainers and consumers. In many software development environments, it is common practice to copy and paste existing source code, as this can significantly reduce programming effort and time. However, if the copied code contains a bug or vulnerability, and the developers copied the code without fixing the bug, they may bring the bug into their own project. Library reuse is a special case in which developers either include the source code of a certain library into their project, or statically link directly to the library. In either way, the bug contained in the copied code will be brought into the new project. Code reuse detection can help identify such bugs resulting from shared source code.

Last but not least, code reuse detection can be applied in numerous scenarios such as software plagiarism detection, open source project license violation detection and binary diffing.

Code reuse detection can be achieved by calculating the similarity of two code regions. The higher the similarity, the more likely these code regions are from the same source code base. In this paper, we present an approach for measuring the similarity of two assembly functions. In particular, our contributions can be summarized as follows:

- We propose a fuzzy matching approach to compare assembly functions. To address the mutations introduced by compilers, our fuzzy matching algorithm operates at multiple levels, i.e., instruction level, basic block level and structure level.
- We design and implement an effective filtering process to prune the search space when comparing a target function against a vast number of functions. We propose two filters that efficiently rule out functions that are unlikely to be matched to our target function. With the help of this filtering process, we can compare one target function against millions of other functions within seconds.
- We design and implement a fully functioning tool for binary code reuse detection based on the proposed approach. Our extensive experiments show that our tool is fast, accurate, and scalable.
- We introduce many use cases, including patch analysis, malware analysis, and bug search, to demonstrate the efficiency and effectiveness of our approach when applied in real-world scenarios.

2. PROBLEM STATEMENT

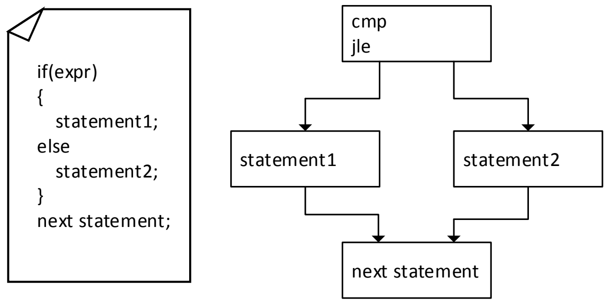
The problem we are trying to solve can be described as follows: Given one target binary function from one executable, and a large repository with thousands or millions of functions from other executables, how to identify all the identical or similar functions from the repository. This problem is two-fold. First, how to compare two assembly functions and obtain a similarity score. Second, how to efficiently retrieve those ones that are likely to be identical or similar to our target function and at the same time, avoid pairwise comparison.

In this work, we establish the similarity of two functions by comparing their control flow graphs (CFGs). The CFG of an assembly function is a directed graph, where nodes represent basic blocks, and edges represent the execution flow between basic blocks.

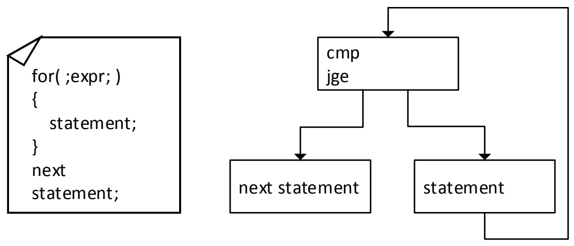
The compiler is responsible for transforming the source code into assembly code. Take C++ for example, generally speaking there are four types of control structures:

- Sequential control structure
- Selection control structure (e.g., if, if-else or switch statement)
- Iteration control structure (e.g., for, while or do-while loop)
- Goto structure

Normally the sequential control structure will not bring addition edges or branches into the control flow graph, while the later three structures would. Figure 1 shows some typical examples of those structures and their corresponding CFGs. Note that as these structures can be nested in source code, so do their corresponding CFGs.



(a) The "if-else" structure



(b) The "for" loop structure

Figure 1: Examples of control structures and corresponding CFGs

Although different compilation environments would bring some mutations or "noise" into the CFGs, still the overall structure is relatively stable. As can be seen from Figure 1, the mapping between source code statements and basic blocks is stable as well.

Based on these observations, we choose to use a basic block-centric approach when comparing two functions. We first find the mapping of basic blocks between these two functions and then for every matching basic block pair, we obtain a matching score. Finally, we calculate the similarity score of two functions from the matching results of the basic blocks.

In this paper, we introduce BinSequence, which uses the similarity score between basic blocks as the building block. Guided by these scores, we continue to find the mapping of basic blocks based on the control flow graph. Finally, we calculate the similarity score from the found mapping.

The rest of the paper is organized as follows. In section 3 we introduce the fuzzy matching approach we use to compare functions. Section 4 provides a detailed description of our filtering process. In Section 5 we evaluate our approach with extensive experiments and give the results. We then review related works in Section 6, and give limitations in section 7. We conclude the paper in Section 8.

3. ALGORITHM DESCRIPTION

Figure 2 depicts an overview of BinSequence framework. First, a collection of interesting binaries such as previously analyzed malware or open source projects that may have been reused, is disassembled. The output is a set of func-

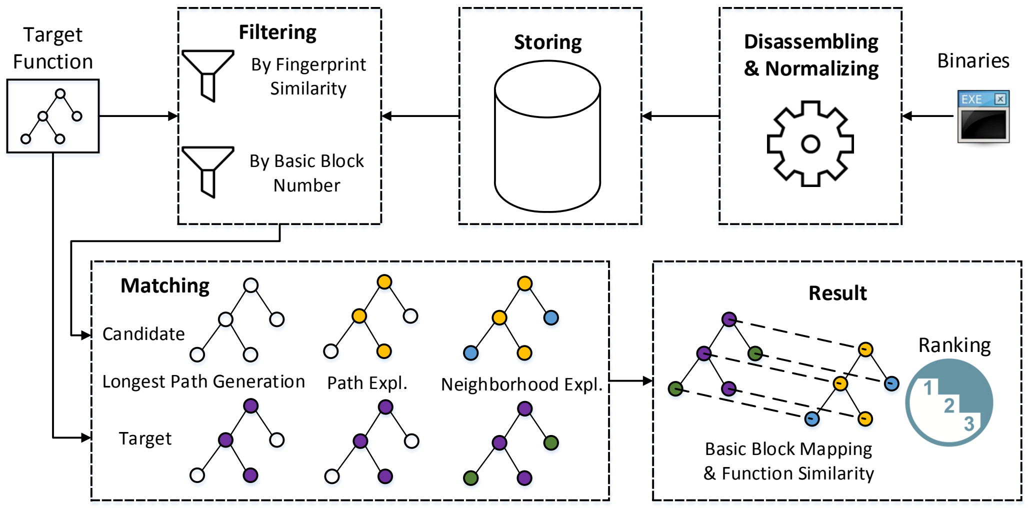


Figure 2: Workflow of BinSequence

tions. We then keep all the functions in a large repository after normalizing them. Given a target function, the naive way is to compare it with every function in the repository and rank the results. However, this is not efficient as most of the functions in the repository are not similar to our target and should thus not be compared. To speed up the process, we focus only on those functions that are likely to be similar with our target. To this end, we adopt a filtering process in which we use two filters. The first filter is based on the number of basic blocks, while the second is based on the similarity of feature sets that we extracted as fingerprints for every function. The output of the filtering process is a subset of functions from the repository, which we call the candidate set. We then perform pairwise comparisons of the target function with every function in the candidate set. The comparison consists of three phases. First, we generate the longest path of the target function. Then we explore the reference function in the candidate set to find the corresponding matching path, from which we can obtain the initial mapping of basic blocks. We then improve the mapping through neighborhood exploration in both the target and reference functions. The output is the mapping of basic blocks and the similarity score of these two functions. After we have done this to every function in the candidate set, we obtain a ranking of functions based on the similarity score.

3.1 Disassembly and Normalization

Given a collection of binaries, the first step is to disassemble each binary to a set of functions. In our experiments, we use IDA Pro [5] as our front-end to perform code analysis and to generate the control flow graph for every function. Since the compiler has many choices with regard to mnemonics, registers and memory allocations when generating the assembly code, it is essential that every assembly instruction in the basic block is normalized before comparison [26].

Note that for most architectures like X84 and X86-64 an assembly instruction consists of a mnemonic and a sequence of up to 3 operands. When normalizing instructions, we keep the mnemonics untouched, and only normalize the operands. We classify the operands into three categories, namely registers, memory references and immediate values. For immediate values, we further normalize them into two categories, memory offsets (addresses) and constant values. The reason to differentiate between addresses and constant values is that addresses would change according to different assembly code layouts while constant values do not. If an immediate value is classified as constant value, we keep the literal value. The motivation is that normally constants stay the same even when different compilers or optimization levels have been used. Some literatures also consider strings as a special type of data constants [12, 17]. In [12] David and Yahav replace an offset with the string and take the string into comparison if the offset points to a string. However, we consider only integers. The reason is that most strings come directly from source code and thus can be modified without difficulty. A malware author could easily evade those string based detection techniques by changing the strings inside the source code without changing the functionality. However, the integers are more related to the functionality, which makes them a better target in reverse engineering.

3.2 Instruction Comparison

Inspired by the recent work in [12], we use a similar strategy when comparing instructions. As depicted in Algorithm 1, for two normalized instructions, if they have different mnemonics, then their matching score is 0 regardless of their operands. Otherwise, we give them a score for identical mnemonic and continue to compare their operands. If their corresponding operands are the same after normalization, then we give them an additional score for each matching operand. Notice that mnemonics represent the low-level machine operations and carry more information than operands, thus we should give a higher score to identical mnemonic. At the same time, to avoid the information carried by operands from getting neglected, this score could not be overly high. Constants also carry much information from the source. When comparing two constant operands, we further compare their literal values. If their literal values are the same, we then give them an additional score.

During our experiments we found that it is appropriate to give score 1, 2 and 3 to identical operand, mnemonic and constant respectively. Using these score values, we can allow those important parts of instructions to match, and at the same time, without getting misled by this biased score strategy. Following this strategy, we can calculate that, the score of comparing `push eax` with `push ebx` is 3, as both are `push REG` after normalization, while the score of comparing `push 0` with `push 1` is only 2 as the literal value of their operands is not the same.

Algorithm 1: Compare two instructions

```
   Input: Two normalized instructions
   Output: The matching score of two instructions
 1 Function CompIns(ins1,ins2)
 2     score = 0
 3     if ins1.Mnemonic == ins2.Mnemonic then
 4         n = num_of_operands(ins1)
 5         score += IDENTICAL_MNEMONIC_SCORE
 6         for $i = 0; i < n; ++i$ do
 7             if operand(ins1)[i] == operand(ins2)[i] then
 8                 if type(operand(ins1)[i]) == CONSTANTS then
 9                     score += IDENTICAL_CONSTANT_SCORE
10                 else
11                     score += IDENTICAL_OPERAND_SCORE
12                 end
13             end
14         end
15     else
16         score = 0
17     end
18     return score
19 end
```

Instead of comparing original instructions, we choose to compare the normalized instructions. The first advantage is more resistance to register reassignment, which is very common in compiler optimization. Second, we want to do a fuzzy matching. This is different from what David and Yahav did in [12], where they use exact matching when comparing operands. Besides, we allow partial matching. For example, we give a score of 5 to instruction pair `cmp [eax],0` and `cmp ebx, 0`, although they are two types of instructions. The first instruction is comparing an immediate value with a memory reference while the second with an register. The reason for allowing partial matching is that, even for the same variable, compilers have the freedom to represent it as a register variable or a memory variable. Allowing partial matching can tolerate these differences.

3.3 Basic Block Comparison

We leverage the longest common subsequence (LCS) method of dynamic programming [11] to compare two basic blocks. The LCS problem is to find the longest subsequence which is common to both sequences. Note that a basic block is also a sequence of assembly instructions. We then leverage the LCS to calculate the similarity score of two basic blocks. We consider every instruction as a letter and use the score strategy presented in Algorithm 1 to obtain the matching score. Notice that we do not draw any conclusion about whether these two basic blocks are identical or should be matched according to this score. Unlike the work in [12], we just use the similarity score as a guide for later use. As shown in Algorithm 2, the output is the largest similarity score that these two basic blocks can achieve with respect to the score strategy we are using. By backtracking the memoization table, we can also obtain the mapping of instructions between this two basic blocks. Some literatures such as [12] also denote this process of leveraging dynamic programming to obtain the mapping, as "alignment". After this "alignment", instructions that cannot be matched can be jumped over. This jumping over instructions is our fuzzy matching at the basic block level. However, for now this mapping is of no interest to us, as we only need the maximum similarity score. Note that there may be different mappings that give us the same maximum score, however, the maximum score is unique. In our algorithm, it is always in the last cell of the memoization table.

A special case is to use Algorithm 2 to compare a basic block with itself. No doubt that the highest score will be achieved only when every instruction is mapped to itself. We define that score as the "self" score of that basic block. Intuitively, this score can be used to measure the information that a basic block carries. A large basic block results in a high self score.

Algorithm 2: Calculate the similarity score of two basic blocks

```
   Input: Two basic blocks $BB1, BB2$
   Output: The similarity score of two basic blocks
   /* M: the memoization table                     */
 1 Algorithm CompBBs($BB1, BB2$)
 2     M = InitTable($|BB1| + 1, |BB2| + 1$)
 3     for $i = 1; i <= |BB1|; ++i$ do
 4         for $j = 1; j <= |BB2|; ++j$ do
 5             $M[i,j] = Max($
 6             $CompIns(BB1[i], BB2[j]) + M[i-1, j-1]$,
 7             $M[i-1, j]$,
 8             $M[i, j-1])$
 9         end
10     end
11     return $M[|BB1|, |BB2|]$
12 end
```

3.4 Longest Path Generation

We have explained how to compare two basic blocks. For every basic block pair, we can obtain a similarity score. The larger the score, the more similar these two basic blocks are. However, this score is derived from the assembly code only, and is thus not sufficient. For example, for one target basic block, we might find multiple basic blocks that have the same similarity score with it. Even worse, we may end up matching it with a wrong basic block simply because its assembly code is more similar to the target by chance.

Inspired by the recent work in [19], we realize that path in the CFG is a robust feature, since path can record every selection the execution flow took when a branch is encountered, and one path represents one complete particular execution. Notice that the functionality of one path is spread across consisting nodes (basic blocks). If we succeed in finding two paths that are equivalent in terms of functionality, it would be trivial to further match their nodes. Again, we can treat the problem of finding matching nodes as an alignment problem where dynamic programming can be applied. Intuitively, one short path does not carry as much information as a long path. Besides, the longer the path, the more matching nodes we could obtain by aligning it with its matching path, which improves both the accuracy and efficiency of neighborhood exploration process (Section 3.6). Thus, we choose the longest path. We use depth first search to traverse the CFG, and then choose the path with the largest number of nodes.

3.5 Path Exploration

After we obtained the longest path of the target function, the next step is to explore the reference function, to try to find the best match of that path in the reference function. We adopt the approach in [19] to do the exploration. In [19] Luo *et al.* used a breadth-first search combined with dynamic programming to compute the highest score of longest common subsequence of semantically equivalent basic blocks. In our case, we leveraged their algorithm to find the corresponding path which has the largest similarity score based on Algorithm 2.

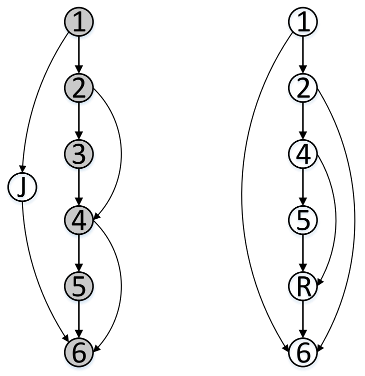
The algorithm for path exploration is similar to the common dynamic programming for computing the LCS of two strings. Since a path is also a sequence of basic blocks, we can treat every basic block as a letter and use the Algorithm 2 as our score strategy. However, there are two differences. First, the length of a string is constant, thus when computing the LCS of two strings the length of the memoization table is also fixed. In path exploration, however, we do not know the length of the memoization table in advance, so we set the initial length to one (Line 2) and add more rows on the run (Line 9). Second, the letters in a given string are sequential; every previous letter has at most one letter following it while a node in a CFG may have multiple successors. That is why we need to combine breadth-first search with the original dynamic programming.

We modified the algorithm in [19] to fit our needs. Given a longest path P from the target function and the CFG G of the reference function, we always start from the head node of G (Line 5). At the beginning of each iteration, we pop out a node from the working queue Q as the current node (line 7). Then we add a new row to the memoization table δ and update the table correspondingly using function LCS (Line 10). It is worth noting that when comparing the current node with every node in path P, we require them to have the same in-degree and out-degree to be matched (Line 22). Otherwise we do not allow them to match by giving them a score of 0 (Line 25). The motivation is that we want to quickly match the "skeleton" of the CFG first. If we failed to match some nodes whose in-degree or out-degree have been changed, we can leave them to the next step, neighborhood exploration. Also note that because of the complexity of the CFG, there might be multiple paths that can lead the execution flow to a certain node. To improve the efficiency, it is important to reduce the search space and prune the unprofitable path. To this end, we use an array σ to store the largest similarity score that we have achieved so far for each node. Every time after updating the table δ for certain node, we continue to compare the obtained new score with the largest score stored in σ (Line 11). If the new score is larger, we then update σ and insert every successor of this node to our working queue Q. Otherwise we do not further explore its successors. The algorithm terminates after Q is empty. The output is the memoization table δ.

Algorithm 3: Path exploration

```
   Input: P: the longest path from the target function,
          G: the CFG of the reference function
   Output: δ: The memoization table
   /* σ: the array that stores the largest LCS
      score for every node in G                          */
1  Function PathExploration(P,G)
2      δ = InitTable(1, |P| + 1)
3      σ = InitArray(|G|)
4      Q = InitQueue()
5      Q.pushback(G_1) //always start from the head node
6      while Q is not empty do
7          currNode = Q.front()
8          Q.pop_front()
9          δ.AddNewRow() //always add a new row to δ
10         LCS(currNode,P) //compare currNode with
            every node in P and update the table δ
11         if σ(currNode) < δ(currNode, |P|) then
12             σ(currNode) = δ(currNode, |P|)
13             for each successor s of currNode do
14                 Q.pushback(s)
15             end
16         end
17     end
18     return δ
19 end
20 Function LCS(u,P)
21     for each node v of P do
22         if SameDegree(u,v) then
23             sim = CompBB(u, v)
24         else
25             sim = 0
26         end
27         δ(u, v) = Max(
28         δ(parent(u), parent(v)) + sim,
29         δ(parent(u), v),
30         δ(u, parent(v)))
31     end
32 end
```

Figure 3 presents an example. Figure 3a shows two simplified CFGs of two functions from open source projects; the grey nodes denote the longest path we found in the target CFG. These two functions are from the same source code. However due to the noise introduced by the compiler, their structures are not isomorphism. Basic block J in the target function consists of one "JMP" instruction, directing the execution flow to the tail block "6". Basic block 3 in the target CFG modifies the value of a local variable in the stack. As can be seen from Figure 3a, basic block 3 does not have a corresponding basic block in the reference CFG because the compiler used the register "ESI" to represent this variable in the reference function. Moreover, the reference CFG has one more basic block R, that restores the original value of "ESI", and then directs the execution flow to the tail. All these changes are very common.



(a) The CFGs of two versions of the same functions and the grey nodes represent the longest path in the target CFG

	v	1	2	3	4	5	6
u	0	0	0	0	0	0	0
1	0	↖ 1	← 1	← 1	← 1	← 1	← 1
6	0	↑ 1	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2
2	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3
4	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
5	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3
R	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
R	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4

(b) The memoization table

Figure 3: An example of path exploration for two CFGs

To do the path exploration, we first initialize the memoization table δ and array σ. Then we insert the head node 1 of the candidate CFG to the working queue Q. We compare node 1 with path P using the function LCS in Algorithm 3 and update the memoization table correspondingly. Notice that here for the purpose of simplicity, we assume that the matching score is either 1 or 0, while a true match has a score of 1, otherwise 0. Since node 1 has two successors, node 6 and 2, we insert them into Q and continue the exploration. Assume we visit node 6 first, then node 2. Node 6 has no successor, we then update the table δ for node 6, and continue to work on node 2. Node 2 has two successors, node 6 and node 4. We also insert them into our working queue. We first work on node 6. Note that this is the second time we insert node 6 into Q. The first time its parent node is 1, and the corresponding partial path is “1→6”, this time its parent node is 2 and the partial path is “1→2→6”. We allow the same node to be inserted into Q as long as they represent different execution paths. Node 6 has no successor. After we finish comparing node 6 with every node in path P, the working queue Q has only one element: node 4. We then work on node 4. It is worth noting that although node 4 in the reference CFG has a corresponding node (node 4) in the path P, the in-degrees of these two nodes are different. Thus, we give them a matching score of 0. Then we put the successors of node 4 into Q. Now Q has two elements, node 5, and node R (parent is node 4). We visit node 5 first, and put its successor node R into Q. Now Q has two elements, node R from node 4 (partial path “1→2→4→R”) and node R from node 5 (partial path “1→2→4→5→R”). Both elements will lead us to node 6, but with different LCS scores. The one from node 4 (complete path “1→2→4→R→6”) will have a final score of 3 while the one from node 5 (complete path “1→2→4→5→R→6”) gives us a score of 4.

We can then backtrack the memoization table δ to get the corresponding path that has the largest sum of similarity score with the target longest path. However, during our experiments we found that considering only the sum of the similarity score may sometimes give undesirable results. We might wrongly match the target path with a long path in the reference CFG. So we decided to normalize the similarity score by taking the target and the found path into consideration. Recall that a path is a sequence of basic blocks, and the self score of one basic block b can be calculated as $\mathit{CompBB}(b, b)$ using Algorithm 2. Then the self score of a path is the sum of self scores of all the consisting basic blocks. We then normalize the score between the target path P and the found path P_f using the following equation:

$$\mathit{NormScore}(P, P_f) = \frac{\mathit{LCSScore}}{\mathit{Score}(P) + \mathit{Score}(P_f)}$$

where the $\mathit{LCSScore}$ is the score obtained from the memoization table δ and $\mathit{Score}()$ is a function that returns the self score of the given path.

We then choose the path with the highest normalized score. By backtracking the memoization table δ, we can obtain a mapping of basic blocks. In the example shown in Figure 3 we can obtain 4 matching basic block pairs: basic block 1 with 1, 2 with 2, 5 with 5 and 6 with 6 in the target and reference, respectively.

3.6 Neighborhood Exploration

While we can continue to extract more paths from the target function and match them in the reference function, this is not efficient. First, the path exploration process takes time. Besides, when we explore certain target path in the reference function, some of the basic blocks may have already been matched in previous paths and we cannot gain much by rematching them. Inspired by the work in [24], we decided to use a greedy, localized fuzzy matching approach to extend the existing mapping. Because we already have all the mappings from path exploration of the longest path, there is a high chance that we can find the correct basic block mapping between two functions.

We first put every matching basic block pair obtained from path exploration into a priority queue based on their similarity score. Then we choose the pair on the top, namely the pair with the largest similarity score as our starting point to initialize the search. We then explore the neighbors of the chosen basic block pair. Note that for every basic block pair in the queue, the two basic blocks have the same in-degree or out-degree. We first consider the successors of these two basic blocks if they have the same out-degree. If they both have only one successor, then we match their successors directly, unless it is inconsistent with the mapping we already have. If they both have more than one successor, then we leverage the Hungarian algorithm [21] to find the best mapping between the two sets of successors that maximize the sum of the similarity score. Similarly, if the found mapping is inconsistent with the mapping we already have, we discard the corresponding match but continue to check other successors. We then do the same to their predecessors if they have the same in-degree.

It is important to note that for those found mapping pairs, the corresponding basic blocks in the pair do not necessarily have the same in-degree or out-degree. If they have the same in-degree, we put them into the priority queue but only explore their predecessors later, when they become the element with the highest priority (similarity score) in the queue. If they have the same out-degree, we explore their successors. If neither their in-degree nor out-degree is the same, we still allow these two basic blocks to be matched, however, we do not put them into the priority queue. In other words, we do not explore their neighborhood, because the likelihood of them being a correct match is relatively lower. By doing this, we achieve a fuzzy matching between the basic blocks of two functions. At the same time, if we mismatched a pair of basic blocks, we still require these two basic blocks to have the same in-degree or out-degree to further examine and match their predecessors or successors. As a result, the error would not propagate. On the other hand, for basic blocks that are correctly matched, we could explore their neighborhood in two directions efficiently.

We continue to do this until the priority queue is empty, i.e., until there is no more neighbors to be explored, or all the neighbors have different in-degree and out-degree and can not be further explored. We then leverage the obtained matching basic block pairs to calculate the similarity between the target function and the reference function.

An assembly function can be looked at as a set of basic blocks, we then calculate the self score of a function by adding the self scores of all the consisting basic blocks. Given two functions, f and g, suppose γ is the set of all the matching basic block pairs we obtained during path exploration and neighborhood exploration, the similarity of these two functions can be calculated as follows:

$$Similarity(f,g) = \frac{2\sum_{\forall(u,v)\in\gamma} CompBB(u,v)}{Score(f) + Score(g)}$$

where u, v are basic blocks, $u \in f$, $v \in g$ and $Score()$ is a function that returns the self score of the given function.

4. FILTERING

We have introduced how to pairwise compare two functions. However, we still need to address the scalability problem, especially when dealing with large data sets. Suppose we have a function repository consisting of one million functions, to find similar functions to a given target function, we have to compare the target function with every function in the repository and rank the results. This is not efficient as a large number of functions are not similar to the target and should not be compared.

To this end, we adopt a heuristic approach to prune the search space by excluding functions that are not likely to be matched. We designed two filters, based on the number of basic blocks and function fingerprint similarity threshold, respectively.

4.1 Filtering By Number of Basic Blocks

The reason to filter by the number of basic blocks is straightforward. It is very unlikely that a function with only one basic block can be matched to another function with one hundred basic blocks. Thus we set a number threshold. If we require two CFGs to be exactly the same, namely isomorphism, then the numbers should also be the same. Since BinSequence performs a fuzzy matching, which allows two structurally-different functions to be matched, the numbers of basic blocks should be allowed to be different. Thus we set up a threshold. This threshold should not be too small, as we may rule out the correct match. On the other hand, the threshold should not be too large, otherwise we can not save much time as not many functions can be ruled out. Assume the threshold is γ, given a target function f, then those functions whose sizes are between $|f| - \gamma$ and $|f| + \gamma$ will pass this filter.

4.2 Filtering By Fingerprint Similarity

The next filter is based on the syntactic property of the code. For every function, we use its normalized instruction set as its fingerprint. More specifically, we use the same technique as introduced in section 3.1 to normalize all the instructions inside a function, to get the normalized instruction set. Given a target function, we then calculate the Jaccard similarity (index) between the fingerprints of the target and every function in the repository. If the Jaccard similarity is above a certain threshold, we then continue to compare the function against the target. Otherwise we simply discard it.

In order to avoid pairwise comparison of fingerprints, we leveraged Minhashing [9] and the banding technique [18]. Minhashing is a technique of using k different hash functions to generate the minhash signature. The banding technique divides the minhash signature into b bands of r rows each. Given a target function, we first generate its fingerprint and the minhash signature of its fingerprint. We divide its minhash signature into b bands of r rows, each. Then the candidate set should be all the functions whose minhash signatures agree in all the rows of at least one band with the signature of the target function. More generally, if we choose n hash functions, b bands, r rows and $n = br$, the Jaccard similarity threshold t imposed by this banding technique is approximately $1/b^{1/r}$ [18].

In general, similar to the filter using number of basic blocks, this filter is lossy as well. Some true matches may have significantly different normalized instruction set, and consequently, fail to pass this filter. To address this problem, in our implementation, we choose b and r so that $t = 1/b^{1/r}$ equals to a relatively low value, e.g., 0.65, so that those functions that are true matches, but with low Jaccard similarity could pass this filter and remain in the candidate set. After using these techniques, to root out all the functions whose Jaccard similarity is above certain threshold, we only need to first choose b and r so that the desired threshold is imposed by the banding technique. and then select all the functions whose minhash signatures agree in all the row of at least one band with the signature of the target function, which can be achieved by one time lookup in the database.

5. EVALUATION

We conducted extensive experiments to evaluate BinSequence in terms of accuracy, performance and scalability. We also performed several experiments on practical scenarios to demonstrate the effectiveness and efficiency of BinSequence when applied to real-world use cases. All experiments were performed on a PC with an Intel Xeon E31220 Quad-Core processor, 16 GB of RAM running Microsoft Windows 7 64-bit.

5.1 Function Reuse Detection

The first experiment is function reuse detection from a large repository. We first try to perform function reuse detection between two versions of the same binary. In this experiment, four different versions of zlib libraries, namely 1.2.5 through 1.2.8 were used. The reason to choose zlib library is it is widely used in many software and operating systems. Since zlib is a well maintained library, we assumed that functions with identical function (symbolic) name across different versions should have the same or similar functionality, and thus, should be matched. We also introduced one group of noise functions, which are all the functions of 1,701 system dynamic library files obtained from Microsoft Windows operating system. The total size of these files including four zlib libraries and 1,701 dynamic library files is around 1 GB and the total number of functions is 2,055,584.

Every time we use the previous version of zlib to match its next version. For example, we first use zlib 1.2.5 as our target set, and put all the functions of its successive version zlib 1.2.6 together with the two million noise functions into database. Then, for every function (with at least four basic blocks) in zlib 1.2.5, we use BinSequence to search for it. Only when the corresponding function in zlib 1.2.6 is ranked first, which means it has the highest similarity, we consider it as a correct match. Otherwise, we consider it as wrongly matched. Afterwards, we do the same process to other versions of zlib.

For all the tests, we used three different fingerprint similarity thresholds: 0.6, 0.65 and 0.7. Those functions whose Jaccard similarity below these thresholds would be ruled out using the techniques explained in Section 4.2. Intuitively as we increase the fingerprint similarity threshold, the number of functions that could pass this filter would decrease. So we choose three different values to thoroughly study its effect. The threshold for the number of basic blocks is set to 35 throughout our experiments.

Table 1a shows the obtained results. Recall that for every target function, we use two filters to obtain a small candidate set from the whole database. The column "Candidate Size" is the sum of the size of the candidate sets for every target function. Intuitively, as we increase the similarity threshold, we end up with a smaller candidate set. As a result, the processing time decreases. We expected that as we increase the fingerprint threshold, the overall accuracy would drop (like zlib 1.2.5 in the table), or at the best would stay the same (like zlib 1.2.6 in the table) because we would get a smaller candidate set and the true match could have been ruled out. The zlib 1.2.7 was a surprise. As we increased the fingerprint threshold from 0.6 to 0.7, the overall accuracy increased from 96.52% to 98.26%. We looked into the reason. When the fingerprint threshold is 0.6, there were two functions, whose true matches did not have the largest similarity; instead, two other functions that happened to have similar code and structure were ranked first. The true matches were ranked #2. As we increased the fingerprint similarity threshold from 0.6 to 0.7, these two functions were ruled out by the filter; as a result, those true matches become the ones with the highest similarity. This also suggest that though our filters are in general lossy, however do not necessarily always decrease the accuracy.

Version	Fingerprint Threshold	Overall Accuracy	Candidate Size	Time (per function)
1.2.5	0.6	96.26%	12346	2.806s
	0.65	94.39%	2727	1.468s
	0.7	91.59%	1911	0.897s
1.2.6	0.6	100%	16315	2.927s
	0.65	100%	2848	1.558s
	0.7	100%	1989	0.913s
1.2.7	0.6	96.52%	16312	2.884s
	0.65	97.39%	2847	1.572s
	0.7	98.26%	1988	0.918s

(a) Function reuse detection between different versions of zlib

Version	Fingerprint Threshold	Overall Accuracy	Candidate Size	Time (per function)
1.2.8	0.6	92.5%	3526	2.204s
	0.65	92.5%	751	1.258s
	0.7	92.5%	242	0.95s

(b) Function reuse detection between zlib and libpng

Table 1: Results for function reuse detection

We also conducted function reuse detection between two different binaries: zlib and libpng. Libpng is a library for processing PNG image format files and is dependent on zlib library. As a result, part of the functions from zlib library are reused by libpng. We first compiled zlib 1.2.8 and libpng 1.6.17 with the debugging information attached. By manually checking both libraries, we identified 40 functions that were user functions in zlib and were reused in libpng. We then used these 40 functions (with at least four basic blocks) in zlib as our target functions, and searched for them in the repository. If the corresponding function in libpng is ranked first, we consider it as a correct match. As shown in Table 1b, we correctly identified 37 reused functions for all the three different fingerprint thresholds, and the overall accuracy was consistently 92.5%.

5.2 Patch Analysis

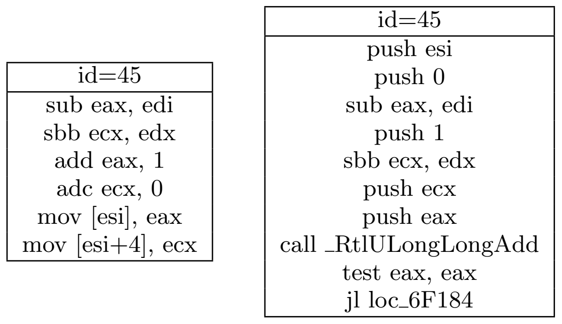
The next experiment is to use BinSequence to recover the vulnerability information. Nowadays as a result of the development of vulnerability mining techniques, more vulnerabilities are being discovered everyday. After a vulnerability is reported to the software vendor, they would release a security patch to fix it often without revealing the detail of the vulnerability or the part of code they have modified to the public. However, by comparing the patched and unpatched versions of the binary, reverse engineers can analyze and understand the vulnerability and the patch within hours. This kind of technique is especially useful for Microsoft's binaries as they release the patch regularly and the patched vulnerability are concentrated in small areas in the binary [23].

We take `MS15-034` [6] as a case study. There is a vulnerability in `HTTP.sys`; when an attacker sends a specially crafted HTTP request to an affected system, the HTTP protocol stack may parse it improperly. As a result, the attacker may execute arbitrary code. Microsoft released a patch `MS15-034` to address this problem. In order to reveal the information of the vulnerability and the patch, we used BinSequence to compare the unpatched and patched versions of `HTTP.sys`. Since the goal is to find out which functions have been patched, we only report functions whose similarity is not 1 after being patched, as similarity 1 means the function remains identical (after normalization) in the patched version.

In total, BinSequence identified 11 functions, whose similarity is not 1 between the patched and unpatched versions. We manually checked these 11 function pairs and found out that 6 functions were actually the same, but were disassembled differently by IDA Pro. Table 2 lists the remaining 5 functions.

Function	Similarity
`UlpParseRange`	0.971783
`UlpBuildSingleRangeMdlChainFromSlices`	0.915530
`UlpBuildMultiRangeMdlChainFromSlices`	0.849870
`UlpDuplicateChunkRange`	0.804167
`UlAdjustRangesToContentSize`	0.501853

Table 2: Patched Functions



```
id=45
sub eax, edi
sbb ecx, edx
add eax, 1
adc ecx, 0
mov [esi], eax
mov [esi+4], ecx
```

```
id=45
push esi
push 0
sub eax, edi
push 1
sbb ecx, edx
push ecx
push eax
call _RtlULongLongAdd
test eax, eax
jl loc_6F184
```

Figure 4: The basic block 45 before and after patching

We then take a closer look at the `UlpParseRange` function. Both the pathched and unpatched versions have 60 basic blocks. BinSequence successfully matched all the basic blocks. Among all these pairs, 59 pairs have a similarity of 1, which means they remain the same after being patched (after normalization). The only changed basic block is basic block number 45.

Figure 4 depicts the basic block 45 before and after the patch. We can see that the patched version is calling a function `_RtlUlongLongAdd` while the unpatched version does not. We can infer that the original function might contain an integer overflow vulnerability. The patched version invokes the `_RtlULongLongAdd` to fix it. Moreover, we can see that the out-degrees of these two basic blocks have been changed. The out-degree of the unpatched is 1 while the patched is 2. Despite this structure change, our fuzzy structure matching approach still succeeded in matching these two basic blocks.

5.3 Malware Analysis

Our next experiment is conducted on two well known malware, Citadel and Zeus. We know that Citedel is derived from Zeus [20]. We also know that Zeus uses RC4 stream cipher function, and Citadel reuses this function with a slight modification [25]. Given the RC4 function in Zeus, our intention is to use BinSequence to identify the reused RC4 function in Citadel.

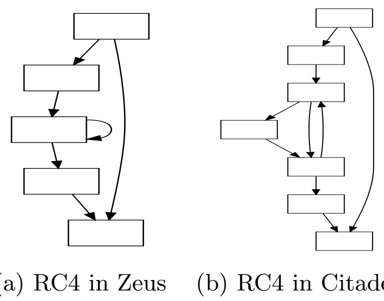
We first disassembled Zeus using IDA Pro and extracted the RC4 function. We then used it as our target function. We also disassembled Citadel and then compared the target function with every function in Citadel, and ranked the results according to the similarity score. Table 3 shows the top 3 functions that have the largest similarity score.

Function	Similarity
`sub_42E92D`	0.689474
`sub_432877`	0.429091
`sub_430829`	0.423913

Table 3: The result of searching RC4 function in Citadel

We manually checked the `sub_42E92D` function, and confirmed that this is the modified RC4 function in Citadel. In total, IDA Pro identified 794 assembly functions in Citadel. That is to say, we successfully identified the modified RC4 function from these 794 functions. Since 794 functions is a relatively small data set, we put the RC4 function of Citadel into those two million functions we used in Section 5.1 and repeated the experiment. Still, BinSequence ranked the modified RC4 function as first, from a repository with more than two million functions.

We also used three binary diffing tools namely BinDiff [1], Diaphora [3] and PatchDiff2 [7] to do this experiment. BinDiff is the de facto standard commercial tool for comparing binary files and both Diaphora and PatchDiff2 are IDA Pro plugins for program diffing. Using the RC4 function in Zeus as target, BinDiff matched it with `Decrypt_String_by_Index` in Citadel, which is not correct. Diaphora reported `sub_40A8B0` with a similarity of 0.22, which is also not correct. We failed to use PatchDiff2 to do this experiment.



(a) RC4 in Zeus (b) RC4 in Citadel

Figure 5: The RC4 function

Figure 5a and Figure 5b show the RC4 function in Zeus and Citadel respectively. Clearly we can see these two CFGs are by no means isomorphism, yet BinSequence ranked the modified RC4 first with the highest similarity. Again, this result demonstrates that our fuzzy matching approach is effective and accurate.

In fact, most of the functions in Zeus have counterparts in Citadel with a higher similarity. More specifically, for 373 (67%) functions in Zeus, Binsequence identified matches in Citadel with a similarity of 1, which means they are exactly identical (after normalization) and 513 (92.1%) functions with a similarity above 0.8. This also confirms that Citadel reused most of the Zeus's functions (functionality). Now when reverse engineering Citadel, the human analyst can focus on those new components and functionality, instead of reanalyzing these reused functions.

5.4 Bug Search

The next experiment is a bug search use case. There is a heap-based buffer overflow in `resize_context_buffers` function in libvpx library used by Firefox [2]. Our intention is to use the `resize_context_buffers` function in libvpx as our target and identify the buggy function in the repository, if there is any. According to the vulnerability data source [2], this bug only exists in Firefox before version 40.0.0. Accordingly, we downloaded different versions of Firefox from 33.0.0 to 40.0.0 directly from the official web site. We only considered the main versions and ignore those subversions.

Firefox	Function	Similarity	Rank
40.0.0	`sub_116D3D02`	0.427699	1/161,932
39.0.0	`sub_1165C97B`	0.657224	1/159,589
38.0.0	`sub_1153BA02`	0.657224	1/155,299
37.0.0	`sub_1155BD63`	0.657224	1/151,416
36.0.0	`sub_115F7CB3`	0.657224	1/152,032
35.0.0	`sub_100CB36B`	0.268199	1/142,304
34.0.0	`sub_101800DA`	0.268199	1/138,329
33.0.0	`sub_108F3DA4`	0.141892	1/135,621

Table 4: Search results for different versions of Firefox

In total, there are 1,196,522 functions in these 8 versions of Firefox, and it took 0.271 seconds for BinSequence to finish the whole comparison. As shown in Table 4, BinSequence uniquely identified the equivalent buggy function in Firefox 36.0.0, 37.0.0, 38.0.0 and 39.0.0. We manually checked the assembly and the source code and confirmed the found functions are indeed the buggy ones. We also checked the `sub_116D3D02` function that has the highest similarity in Firefox 40.0.0, and found that it was actually the patched `resize_context_buffers` function. For Firefox 33.0.0 through 35.0.0, BinSequence found three functions with a relatively low similarity. We found that these three versions of Firefox were using a different version of libvpx. As a result, the buggy function actually does not exist in these three versions. Still, BinSequence reported the ones with the highest similarity in the corresponding project.

We also used BinDiff, Diaphora and PatchDiff2 to do this experiment. More specifically we apply these tools on Firefox 39.0.0 and 40.0.0 since Firefox 39.0.0 contains the equivalent buggy function, and Firefox 40.0.0 has the patched buggy function. BinDiff correctly identified the equivalent buggy function in Firefox 39.0.0 and the patched function in Firefox 40.0.0. However, we noticed that the heuristic that BinDiff used was "string references" and one identical string exists in these functions. If we modify this string a little bit and redo the experiment, BinDiff would fail and wrongly match the buggy function to two other functions. For Firefox 39.0.0 and 40.0.0, both Diaphora and PatchDiff2 failed to match the buggy function or the patched buggy function.

5.5 Function Matching

In this experiment, we compare BinSequence with FCatalog [4], Diaphora [3], PatchDiff2 [7] and BinDiff [1] for function matching. FCatalog performs k-gram analysis and use minhash signatures to compare two functions. All Diaphora, PatchDiff2 and BinDiff are binary diffing tools and can create a mapping of functions between two versions of the same binary. BinSequence, however is not confined to comparing two binaries. In this experiment, given a target function in one binary, we use BinSequence to compare the target with every function in the other binary and match the target to the function with the largest similarity score.

Throughout this experiment, we continue to use zlib 1.2.8. However, we compile it in release mode using two different compilers, namely MSVC 2010 and MSVC 2013. The reason of choosing two compilers is to introduce certain "noise" into the code. We then use the functions in zlib 1.2.8 compiled by MSVC 2010 as our target set, and functions compiled by MSVC 2013 as the candidate set. For every non-empty function (with at least 4 instructions) in the target set, we use BinSequence to find the matching function (with the highest similarity) in the candidate set. In this experiment, all the function names of both binaries are stripped away. But we use the function names in the program debug database as the ground truth to verify if the matching is correct.

Tool	Correctly Matched	Unmatched	Overall Accuracy
FCatalog [4]	5	139	3.47%
Diaphora [3]	105	10	72.92%
PatchDiff2 [7]	110	28	76.39%
BinDiff [1]	130	5	90.28%
BinSequence	135	0	93.75%

Table 5: Comparison with other tools

Since zlib is a relatively small library, all the tools finished the comparison within seconds. Table 5 shows the results. In total, our target set has 144 functions with more than 4 instructions. FCatalog correctly matched 5 functions, but failed to find any match for the remaining 139 functions. Diaphora matched 105 functions. However, for 10 functions, Diaphora failed to match them and categorized them to "Unmatched" group. The overall accuracy for Diaphora is about 72.92%. Similarly, if BinDiff failed to match one function with another, BinDiff would classify it into "Unmatched". As shown in Table 5, there are 5 functions that BinDiff failed to match. However, given one target function, BinSequence simply compares it with every function in the candidate set and match it to the one with the highest similarity. As a result, BinSequence has no "Unmatched" category.

We can see from Table 5 that BinSequence achieves the highest accuracy, 93.75%. The reason is that BinSequence is performing a fuzzy matching, which can better address the mutations introduced by different compilers.

6. RELATED WORKS

A lot of work has been done on the problem of code reuse detection and function similarity calculation. We briefly review related works in this section.

Flake [14] and its extension [13] presented a pioneer work of structural comparison approach, Bindiff. Throughout our work we also leverage the structural information to compare two functions, but we focus more on fuzzy matching. Moreover, BinDiff is mainly used for comparing two different versions of the same binary, while BinSequence is an assembly function-centric code reuse detector.

BinHunt [15] also uses a structural approach. It uses symbolic execution to compare basic blocks and backtracking algorithm to try to find the graph isomorphism between CFGs. However, both techniques are too strict. As a result, Binhunt is suitable for finding semantic differences, not for large scale code reuse detection.

Khoo *et al.* [17] presented a binary code search engine named Rendezvous. In their work, multiple features including mnemonic n-grams, control flow subgraphs and data constants are extracted from the binary functions. However, they do not take the code of the node into account when matching control flow subgraphs. Consequently, the uniqueness of control flow subgraphs is relatively low and a high number of false positives are produced by their approach.

Another approach called SIGMA [8] has been proposed for identifying reused code in binaries. This technique uses a graph-based representation of code, abstracting away much of the instruction-level detail in favor of structural properties of the program. However, the detection algorithm used is time consuming.

Pewny *et al.* [24] proposed TEDEM, which is a binary code reuse detection system which can identify the buggy function from a set of reference functions. Unlike previous semantic-based works which leverage symbolic execution and theorem proving, Pewny *et al.* designed a novel way of comparing code regions semantically by leveraging tree edit distance. However, their approach captures all the functionality and “side effects” of code regions, which makes TEDEM suitable for bug search, but inappropriate for code reuse detection where “noise” is very common.

Ng and Prakash [22] proposed a tool to identify binary code reuse. They Adopted two approaches, semantic execution and syntactic matching to compare two function. The problem is that semantic execution is too strict while syntactic matching on the other hand, is too coarse.

David and Yahav [12] proposed a tracelet-based code search. They first break the CFG into small tracelets and use LCS to align two tracertlets. This is also how BinSequence compares two basic blocks. To deal with register reassignment, they leverage constraint solver to rewrite the assembly code in tracelet. But they did not take the whole structure of the CFG into consideration, and many structure information was lost when they break down the CFG. As a result, they admit their tool produces bad results when applied on functions with less than 100 basic blocks. Besides, their approach is not scalable.

Luo *et al.* [19] combined symbolic execution with longest common subsequence to compare two functions and binaries. The problem is that the overhead of symbolic execution is very high which renders this approach impractical when dealing with a large data set. Also, it suffers from scalability problem.

7. LIMITATIONS

Our approach has the following limitations:

- False positive: BinSequence can compare one target function with a repository of functions. However, when there is not a true match in the repository, BinSequence still produce a function ranking according to similarity scores. It is difficult to determine a threshold, to root out all the function that should be reported.

- Function inlining: If a target function is inlined in another function, then our approach may not match these two functions. However, normally function inlining only happens to small functions. Consequently, their functionality are straightforward, and it does not really help reverse engineers much to search for those small functions from a large function repository. On the other hand, if our target function inlined another small function, then there is a high chance that BinSequence can still match them since we are doing a fuzzy matching.

- Equivalent instructions: The compilers may use different instructions to accomplish the same functionality. For example, `mov eax, 0` and `xor eax, eax` have the same functionality but different mnemonics. However, they will be normalized to different instructions by our approach. Future versions of BinSequence may overcome this limitation by dividing instructions into different classes and let instructions in the same class to be matched.

- Instruction reordering: The compiler may change the order of instructions for alignment or pipelining. When we use LCS to compare two basic blocks, we take the order of their consisting instructions into account. As a result this will decrease the final similarity score a litter bit, but it will not jeopardise the entire ranking. The reason is that the layout of instructions inside one basic block corresponds to the layout of the source code. Besides, some instructions may have dependencies upon previous instructions. Thus, the compilers can not change the order randomly. We can still catch the overall pattern using LCS.

- Basic block reordering: Similarly, basic block reordering decreases the final similarity score because we take the order into consideration when doing path exploration and neighborhood exploration. However, the overall layout of basic blocks still corresponds to the source code. Besides, some basic blocks might have some dependencies upon previous basic blocks, as a result, their orders cannot be changed. Thus, our fuzzy matching approach can still capture these patterns.

- Obfuscation: When designing BinSequence, we assume that the binary is unobfuscated. In other words, BinSequence mainly deals with unobfuscated code. However, we also evaluated the robustness of BinSequence on obfuscated code. We used obfuscator-llvm [16] as our obfuscator, and experimented with three obfuscation techniques: bogus control flow, control flow flattening and instructions substitution. In our test, the accuracy of BinSequence after these three obfuscations have been applied is 53.01%, 3.61% and 93.98%, respectively. We can see that control flow flattening is a challenge for BinSequence since we the structure of CFGs into consideration. Future work may involve merging basic blocks, or spiliting one basic into multiple basic blocks, to achieve a better matching for this case. Besides, deobfuscators could be used as a front-end, to help analyze the obfuscated code.

8. CONCLUSION

In this paper we presented a fast, accurate and scalable binary code reuse detection system named BinSequence. Unlike previous literatures, we focus on fuzzy matching that operates at instruction level, basic block level and control flow structure level. To enable BinSequence on large data sets, we designed two filters to save analysis effort by ruling out functions that are not likely to be matched.We conducted extensive experiments and our results strongly suggest that BinSequence can achieve high performance function ranking.

We also applied BinSequence on many practical use cases. By leveraging BinSequence on both patched and unpatched executables, we succeeded in revealing the vulnerability and the patch information. By performing function reuse detection, we managed to identify the reused RC4 function in two real-world malware, Zeus and Citadel. We also successfully identified the buggy function in various versions of Firefox. We believe that BinSequence can be of great help in many reverse engineering and security scenarios.

Acknowledgment

The research leading to this work is supported by a research grant under the Department of National Defence/Natural Sciences and Engineering Research Council of Canada (NSERC) grant in collaboration with Google.

9. REFERENCES

[1] BinDiff. http://www.zynamics.com/bindiff.html.

[2] CVE-2015-4485. http://www.cvedetails.com/cve/CVE-2015-4485/.

[3] Diaphora: A Program Diffing Plugin for IDA Pro. Available at: https://github.com/joxeankoret/diaphora.

[4] FCatalog. http://www.xorpd.net/pages/fcatalog.html.

[5] IDA Pro. https://www.hex-rays.com/products/ida/.

[6] MS15-034. https://technet.microsoft.com/en-us/library/security/ms15-034.aspx.

[7] PatchDiff2: Binary Diffing Plugin for IDA. Available at: https://code.google.com/p/patchdiff2/.

[8] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[9] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, 2006. FOCS'06*, pages 459–468, 2006.

[10] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi. skywiper (aka flame aka flamer): A complex malware for targeted attacks. *CrySyS Lab Technical Report*, 2012.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 2009.

[12] Y. David and E. Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–360, 2014.

[13] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[14] H. Flake. Structural comparison of executable objects. In *Proceedings of the IEEE Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 161–173, 2004.

[15] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Proceedings of the 10th International Conference on Information and Communications Security*, pages 238–255, 2008.

[16] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM – software protection for the masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15*, pages 3–9, 2015.

[17] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338, 2013.

[18] Leskovec, Jure and Rajaraman, Anand and Ullman, Jeffrey D. *Mining of Massive Datasets.* Cambridge University Press, 2014.

[19] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

[20] J. Milletary. Citadel trojan malware analysis. *Dell SecureWorks*, 2012.

[21] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[22] B. H. Ng and A. Prakash. Exposé: discovering potential binary code re-use. In *Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 492–501, 2013.

[23] J. Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat technical Security Conference*, 2009.

[24] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[25] A. Rahimian, R. Ziarati, S. Preda, and M. Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.

[26] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the 18th international symposium on Software testing and analysis*, pages 117–128, 2009.