Measuring Similarity of Binary Programs using Hungarian Algorithm

Yeongcheol Kim
Dept. of Computer Sci. and Engineering
Chungnam National University
Daejeon, South Korea
aree91@cnu.ac.kr

Seokwoo Choi
National Security Research Institute
Daejeon, South Korea
seogu.choi@gmail.com

Eun-Sun Cho
Dept. of Computer Sci. and Engineering
Chungnam National University
Daejeon, South Korea
eschough@cnu.ac.kr

Abstract—**We introduce a similarity matrix based on n-grams and longest common sequences extracted from pairs of functions. Devising an adapted Hungarian algorithm for such features, we envision to get reasonable results for similarity analysis on different versions of the same functions.**

I. Introduction

Binary program analysis is essential for malware detection, vulnerable code detection, code theft detection, and so on. However, due to the complexity of analyses, analyzing binary programs often costs too much time to make a proper reaction in time. To solve this problem, one would overlook some portion of codes which seem less critical. But in this way, important codes are also likely to be missed, since it is not trivial to figure out less critical portion of binary codes for specific goals, usually resulting in inaccuracy of analysis.

In this paper, we aim to spot the portion of codes which would be safely skipped, suited for fast malware detection. For this purpose, we try to notice the binaries of well-known functions which are safe in the sense of malicious behaviours, so as to enable analyzing suspicious codes effectively by excluding spotted functions in the analysis process.

We could scan a whole binary program to compare the byte string or the hash value with those of probably safe functions which might be embodied in a suspicious program. However, we should consider several factors that can change the binary codes slightly, such as the version of the function, the versions of libraries used in the program, and the sort of the compiler and the options used to build the binaries from the source codes. Therefore a program with the same semantics may have differences in the binary images, so other than exact matching based on the byte strings or the hashes, we need to introduce similarity measuring methods between two binary functions.

There have suggested several solutions to measure and determine similarity of binary codes, based on the cosine values or Longest Common Subsequences (LCS) of two functions, the comparison results of important features like n-grams and semantic abstractions of functions, and so on. But most of them are for different purposes such as code theft detection and patch detection. Note that over-estimated simiarity in those cases is not so harmful as in ours; skipping safe-looking malicious codes (that is, looking similar to well-known safe functions) would yield wrong results in malware detection.

[1] proposed a method of measuring the similarity by applying the Longest Common Subsequence (LCS) algorithm between mnemonics of two functions. However, if the order of the basic blocks in the CFG constituting the function is modified, which entails that the order of the mnemonics is also modified, so the length of the LCS becomes small and the similarity value is lowered (that is, similarity is under-estimated.)

[2] introduces a similarity measurement that calculates the edit distance between Control Flow Graphs (CFGs) of both functions. Before comparing CFGs of two functions, it calculates the similarity of basic blocks constituting each CFG. However, based on graph edit distances and Hungarian algorithm, it suffers from huge overhead of extraction and complex comparison process of the large amount of information, so not appropriate to be used to save time for other analysis like malware detection. Similarly, other approaches [4], [3], mainly focusing on accuracy, also suffer from overhead of calculation, not acceptable for spotting safe functions in malware analysis.

In this paper, we propose a similarity measuring method, which performs sufficiently fast so as to be used as a helper tool for other analysis, while still works well when two functions do not exactly match. First we extract the n-grams from the two functions to be compared, generate the matrix by applying LCS between the n-grams, and find the matching relationship between the most similar n-grams in the matrix through Hungarian algorithm. The final goal of this method is to support resillience against slight changes in the order of basic blocks or instructions due to compilers, compile options and versions, while taking into account semantics of the function including the instruction execution flow and the instructions constituting the function.

II. Background

A. n-gram and Longest Common Subsequence (LCS)

n-gram is a contiguous sequence of n items that can be obtained from a given sequence of anything such as text or speech. When we use n-grams of the instruction lists of two binaries to be compared, it allows more flexibility than using intstruction lists; the pair of whole sequences of mnemonics need not be the same, for high similarity. For instance, applying the 3-gram method to function A is as follows;

Mnemonics of Function A : [mov, mov, cmp, jnz, push, call, pop, pop, ret]
n-gram set of Function A = {{mov, mov, cmp}, {mov, cmp, jnz} ,{cmp, jnz, push}, {jnz, push, call}, {push, call, pop}, {call, pop, pop}, {pop, pop, ret}}

LCS is the longest subsequence that a given two sequences have in common. The length of the LCS of the mnemonic sequences of two functions can be used as a simple measure of similarity [1]. For instance, LCS of function A and function B is as follows;

Mnemonics of Function A : [mov, mov, cmp, jnz, push, call, pop, pop, ret]
Mnemonics of Function B : [mov, mov, xor, cmp, je, call, pop, pop, ret]
LCS between A and B : [mov, mov, cmp, call, pop, pop, ret]

B. *Hungarian algorithm*

The Hungarian algorithm is an algorithm to solve the $n \times n$ matching problem. Although the time complexity of the solution of the $n \times n$ matching problem is $O(n!)$, Hungarian algorithm, an approximated solution, reduces it to $O(n^3)$. In [2], they use Hungarian algorithm to find the best matching cost from the CFG edit distance matrix, which means the CFG similarity between two functions. That is, after a matrix having edit cost between the nodes constituting the CFG as an element is generated, the minimum edit cost between the two graphs is calculated by applying this algorithm. The limitation of this approach is that $O(n^3)$ is still time consuming when it comes to large amount of data related to CFGs they used.

Instead of CFGs, we extract two n-gram sets of mnemonic sequences of two functions, apply LCS to the pairs of n-grams from two n-gram sets to generate a matrix having the number of sequences in common as elements, and measure the similarity of two functions by applying Hungarian algorithm. Additionally, we also propose a method to reduce false positives by using indices of matrix computed through Hungarian algorithm.

III. Proposed Method Using n-gram & Hungarian Algorithm

We use n-gram, LCS, and Hungarian algorithm to measure the similarity between two functions. The following is the process of measuring similarity;

Step 1) After generating an n-gram set from the instructions that constitute the function, a matrix is generated with the length of the LCS calculated by applying LCS algorithm to the elements between the sets.

Step 2) Apply Hungarian algorithm to the generated matrix, and calculate the indices of the matrix that constitute the largest sum of elements that do not select rows and columns as duplicates.

Step 3) Detect the same function more precisely using indices calculated through Method I and Method II, as below;

Method I: Using counts of indices on the relation of $y = x$

As shown in Figure 1, the larger the number of indices with a slope of 1 from the coordinates (0,0) in the indices of the matrix calculated by Hungarian algorithm, the higher the probability that the two functions to compare are the same. The formula for calculating the similarity using the $n \times n$ matrix described in this part is as follows.

$$similarity = \frac{\#\ of\ a\ slope\ between\ index\ and\ (0,0)\ is\ 1}{n}$$

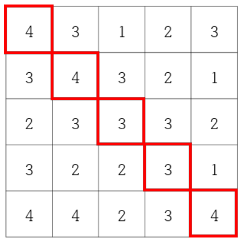
Calculating the similarity of the Figure 1 according to the formula, it is measured as $5/5 = 1$.

Method II: Using frequencies of consecutive indices keeping the slope 1 between indices

Given the indices of the matrix computed by Hungarian algorithm as shown in Figure 2, the higher the frequency that the slope between the indices becomes successively 1, the higher the probability that the two functions to compare are the same. When the $n \times n$ matrix and m indices maintain a slope of 1 continuously, the similarity formula for the method described in this part is as follows.

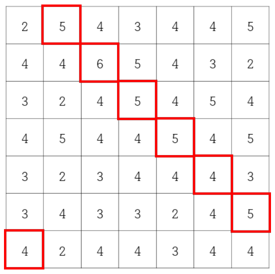
$$similarity = \frac{\#\ of\ slope\ of\ m\ consequtive\ indices\ is\ 1}{n - m + 1}$$

When m = 2, the similarity of the Figure 2 is calculated according to the formula, and it is measured as $5/(7-2+1)$ = $5/6$. For the case of m = 3, the similarity is calculated as $4/(7-3+1)$ = $4/5$.



$\boxed{4}$	3	1	2	3
3	$\boxed{4}$	3	2	1
2	3	$\boxed{3}$	3	2
3	2	2	$\boxed{3}$	1
4	4	2	3	$\boxed{4}$

Fig. 1. Example of Method I



2	$\boxed{5}$	4	3	4	4	5
4	4	$\boxed{6}$	5	4	3	2
3	2	4	$\boxed{5}$	4	5	4
4	5	4	4	$\boxed{5}$	4	5
3	2	3	4	4	$\boxed{4}$	3
3	4	3	3	2	4	$\boxed{5}$
$\boxed{4}$	2	4	4	3	4	4

Fig. 2. Example of Method II

IV. Preliminary Experiments and Discussions

We conduct experiments on functions from two different versions (1.0.1f and 1.0.2h) of OpenSSL DLL files, which are assumed safe and often appear in suspicious codes. When we apply only Hungarian algorithm on LCSs of two n-gram sets, but omitting Step 3 without applying method I or II, F-measure (that is, the harmonic mean of precision and recall [5]) is not more than 0.329. However, F-measure gets up to 0.517 when we complete the whole steps from Step 1 to Step 3, applying both method I and method II. We believe that this result shows a good sign of feasibility.

The proposed method reduces the amount of information needed and the analysis time, and also expected to be robust even when the order of mnemonics is modified by block, because considering flow of instructions in a block (by using n-grams) as well as between blocks (by using the slop value of the indices.) We envision that it will effectively save binary analysis time for malware detection.

References

[1] Jong-Cheon Choi, Seong-Je Cho, *Open Source Software Detection based on Opcode k-gram at Binary Code Level,* Journal of KIISE, 2014. 2.

[2] Patrick P.F. Chan and Christian Collberg, *A Method to Evaluate CFG Comparison Algorithms,* Quality Software (QSIC), 2014

[3] Lannan Luo et. al, *Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparsion with Applications to Software Plagiarism Detection,* Quality Software (QSIC), 2014

[4] Yaniv David, Nimord Patrush and Eran Yahav, *Statistical Similarity of Binaries,* ACM SIGPLAN conference on PLDI, 2016

[5] Tiffanry Bao, et. al, *ByteWeightL Learning to Recognize Functions in Binary Code,* USENIX Security Symposium, 2014