Similarity Detection in Binary Images of IoT Devices

Golam Kayas, Fei Zuo, Suhan Jiang
Email: golamkayas@temple.edu, tuh39530@temple.edu, tug67249@temple.edu

***Abstract*—The next wave in the era of computing will be outside the realm of the traditional desktop. In the Internet of Things (IoT) paradigm, we will be surrounded by a cluster of IoT devices. Source code sharing is very pervasive and common in IoT application development. In this project we will try to find a useful method to detect the similarity among the binaries installed in IoT devices and built in the same architecture. Binary code similarity detection has many concrete security applications such as plagiarism detection, vulnerability search and malware detection, etc. Therefore, our research possesses an enormous practical meaning for security issues in IoT devices. Especially, in this project, we analyze the basic blocks which are generated by decomposition and leverage control flow graph (CFG) to represent a module for one single program. After that, we carry out an approximate matching between two CFGs to look up some similarities. Furthermore, in order to enhance the precision of similarity detection, some significant features are extracted from basic blocks, which will help us to confirm the comparison result from CFGs matching. Finally, some experiments are carried out to evaluate the performance of our scheme. It shows that our approach is feasible and applicable in similarity detection of binary images in practice.**

I. Introduction

Nowadays, IoT devices are becoming ubiquitous in peoples lives as a result of the availability of mobile and heterogeneous devices.

In order to reduce the development complexity, some open-sourced libraries (e.g. OpenSSL) or frameworks (e.g. ODK 2.0) are inevitably and massively used by engineers in practice. Some bugs in these libraries or frameworks often make the IoT devices more vulnerable towards potential attacks and security issues[1]. For example, bugs in OpenSSL lead to the problems like HeartBleed attack [2] which can cause a huge damage in the affected devices.

The abundance of IoT devices really opens the doors for new problems such as patching the IoT devices are hazardous, even finding the current binary versions is not easy as well. Using binary comparison will be a great way to find the status of the currently installed binary and take further actions (e.g. patching). For the huge number of IoT devices it is really arduous to install a new patch to fix the well-known bugs. To identify the security issues, before the crucial bug or attack occurs binary comparison is a big step.

More importantly, IoT devices usually have to face the fact that their resources are constrained such as low memory and limited energy supply. As a result, it is impractical to setup a defense scheme such as anti-virus software or firewall in the IoT devices. These properties surely make finding out the potential risks is of importance for IoT devices. Using similarity detection to automatically find out potential vulnerabilities is becoming popular because of the high performance and low cost. For example, based on similarity detection, the authors in [3] found 38 potentially vulnerable images just by looking at top 50 results according to the similarity.

Consider this scenario : Dr. Xs lab is currently using a big number of IoT binaries for the implementation of a mobile computing project. Some of the IoT devices have some source code from OpenSSL. Now Dr. X wants to patch the binaries contain OpenSSL code. They create a batch of binary images aggregated with the same code from OpenSSL for IoT devices. After very tough manual analysis, the researchers successfully labelled some binaries which contains some source code from OpenSSL. However, due to the manual analysis is very inefficient and expensive, Dr. X still has massive images unlabeled. The researchers expected to figure out which images in the remaining group share the common code with their labeled counterparts in an economical way. Consequentially, they need a scheme to automatically detect the similarity between the binaries.

The goal of our project is going to address the problems similar to the aforementioned one. More concretely, our contribution will be:

- Generate the basic blocks from the decompiled code of binary and extract features from them.
- Generate CFG from the decompiled code and compare them with efficient and approximate graph matching algorithms.
- Detect the similarities between two binary images through the features extracted from basic blocks.

II. Related Work

Early attempts to detect the similarities mainly focus the source-code level. Some pattern matching and information retrieval techniques based on token level granularity are utilized to lexically analyze a program. Not only the bag of tokens, but also the sequence of the tokens are taken into account to do the similarity comparison. CCFinder[4] and CP-Miner[5] are two well-known approaches in this category.

In detail, CCFinder utilizes a suffix-tree algorithm to measure the similarity of the sequence of lexical components. This approach is computationally expensive and consumes a large amount of memory. Besides, CP-Miner parses a program and compares the resulting token sequences using the frequent subsequence mining algorithm. Relying on the heuristics, the authors [5] claim their method can well improve the efficiency. However, the mining complexity of CP-Miner is still $O(n^2)$ in the worst case which make CP-Miner cannot handle the cases at scale.

By contrast, SourcerCC [6] utilizes the bag-of-tokes technique to implement the similarity detection based on function level granularity. Similarly, another method named VUDDY which is carried out by function level granularity is proposed in [7] by S. Kim et al. They assert their approach can detect code clones in large programs efficiently based on some filtering techniques.

Unfortunately, all the works mentioned above are based on source code to do the similarity detection. We have to recognize that the application source code in IoT devices are usually very difficult to obtain. J. Pewny et al. [8] explore the method to detect similarities in binaries base on semantics using the expression tree for each basic block. However, the computational complexity of their method are very expensive.

Back to our scheme, we try to analyze the basic blocks and conclude some valuable features to cooperate with the approximate graph matching to enhance detect the similarities. In this way, we can keep both an acceptable computational complexity and a high accuracy.

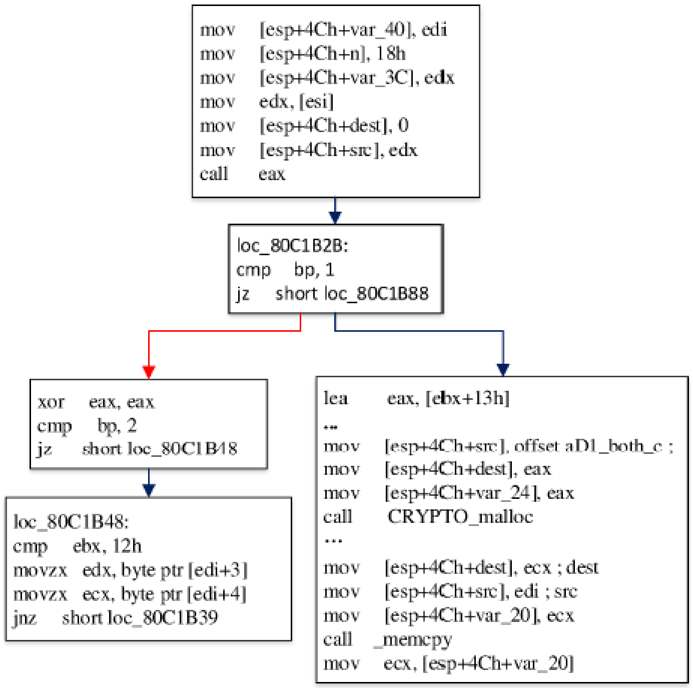
III. Algorithm Design and Implementation

A. CFG Generation

In the first stage, we will take the released binaries and decompile them into the assembly code for the purpose of our analysis. Then we will try to generate the basic blocks of the assembly code generated by decompilation.

IDA Pro [12] is widely acknowledged as the best static disassembly tool currently available for distinguishing code from data in arbitrary binaries [13,14]. It combines straight-line, heuristic and execution emulation-based disassembly while also providing an extensive GUI interface and multiple powerful APIs for interacting with the disassembly data.

In this phase, we generate the CFGs using the basic blocks as nodes. Also, we provide Figure 1 as an example of CFG. More concretely, considering the popularity and the reliability of IDA pro, we choose it as our disassemble tool to analyze binaries.



```
mov     [esp+4Ch+var_40], edi
mov     [esp+4Ch+n], 18h
mov     [esp+4Ch+var_3C], edx
mov     edx, [esi]
mov     [esp+4Ch+dest], 0
mov     [esp+4Ch+src], edx
call    eax
```

```
loc_80C1B2B:
cmp    bp, 1
jz     short loc_80C1B88
```

```
xor     eax, eax
cmp     bp, 2
jz      short loc_80C1B48
```

```
loc_80C1B48:
cmp     ebx, 12h
movzx   edx, byte ptr [edi+3]
movzx   ecx, byte ptr [edi+4]
jnz     short loc_80C1B39
```

```
lea      eax, [ebx+13h]
...
mov     [esp+4Ch+src], offset aD1_both_c ;
mov     [esp+4Ch+dest], eax
mov     [esp+4Ch+var_24], eax
call     CRYPTO_malloc
...
mov     [esp+4Ch+dest], ecx ; dest
mov     [esp+4Ch+src], edi ; src
mov     [esp+4Ch+var_20], ecx
call    _memcpy
mov     ecx, [esp+4Ch+var_20]
```

Figure 1: An Example of Control Flow Graph

We give some sample binaries as input to IDA Pro. Then the "gdl" (graph description file) files which consist

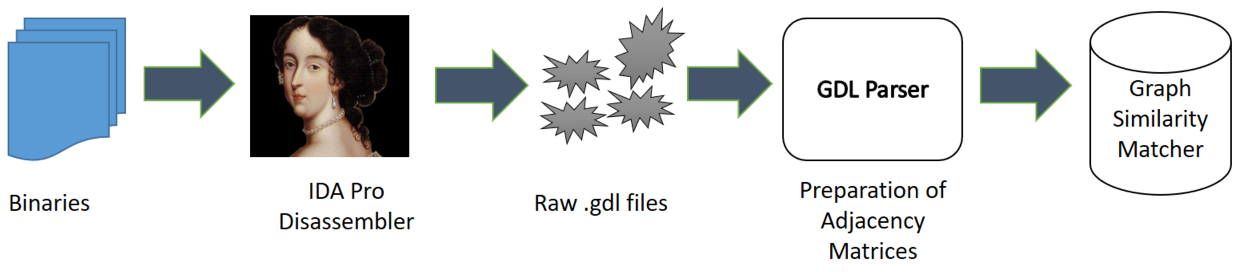


Figure 2: Flowchart of CFG Generation

of the information of the control flow graphs from the binaries will be generated. But we need to generate the properly formatted matrices to represent directed acyclic graphs (DAGs), instead of raw output files from IDA Pro. The formatted matrices will be later feed to the graph matching algorithm (described in subsection B). We developed a Java-based ”GDL parser”, which is capable to read the syntax of the ”gdl” files. The ”GDL parser” automatically extract the information needed to generate formatted adjacency matrices and prepare the input files of the graph similarity matching algorithm. See Figure 2 to get more details of the process of CFG generation from binaries.

B. *Approximate Graph Matching*

Now, we have transformed the original problem to a graph matching problem, specifically in the graph pattern-matching. Graph pattern-matching is a generalization of string matching and two-dimensional pattern-matching that offers a natural framework for the study of matching problems upon multi-dimensional structures. As we all know, it is an NP problem which suffers from its high computational complexity. Thus, we want to use an approximate algorithm to solve this problem. There are several other graph matching algorithms in [9] and [11]. However, we finally decide to choose the algorithm proposed in [10] considering it is very primitive and sufficient to get good matching results.

In general, this algorithm uses neighbor matching to measure the similarity score of the two directed graphs. In detail, we will use two directed graphs, each representing the control flow of the basic blocks in one binary image. Then we want to calculate the similarity between these two directed graphs. In fact, the definition of graph similarity is indeterministic. Here, we want to find the maximum matching between these two directed graphs, then we use ratio between the nodes in the maximum matching graph and the average nodes of these two directed graphs as our similarity degree.

This algorithm can be divided into two periods. In the first step, it iteratively measures the similarity of nodes in the two graphs. Then in the second step it calculates one similarity score using those node similarity measures. Below are some basic ideas about this neighbor matching algorithm.

A directed graph $G = (V, E)$ is defined by its set of nodes V and its set of edges E. There is an edge between two nodes i and j if $(i, j) \in E$. For the edge $e = (i, j)$, the source node is the node i, and the terminating node is the node j.

Given two directed graph G_A and G_B, we firstly define the similarity of graph nodes.

Two nodes $i \in V_A$ and $j \in V_B$ are considered to be similar if neighbor nodes of i can be matched to similar neighbor nodes of j.

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}$$

This equation will calculate the similarity of the ith node of G_A and the jth node of G_B in the $(k+1)$th iteration. As we see, we need to calculate $s_{in}(i,j)$ and $s_{out}(i,j)$ in $(k+1)$th iteration first. These $s_{in}(i,j)$ is the in degree similarity of node i in G_A and j in G_B. $s_{out}(i,j)$ is the out degree similarity of node i in G_A and j in G_B. They can be calculate using following equations.

$$s_{in}^{k+1}(i,j) \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x^k_{f_{ij}^{in}(l) g_{ij}^{in}(l)}$$

$$s_{out}^{k+1}(i,j) \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x^k_{f_{ij}^{out}(l) g_{ij}^{out}(l)}$$

$$m_{in} = max\left(id\left(i\right), id\left(j\right)\right)$$
$$m_{out} = max\left(od\left(i\right), od\left(j\right)\right)$$
$$n_{in} = min\left(id\left(i\right), id\left(j\right)\right)$$
$$n_{out} = min\left(od\left(i\right), od\left(j\right)\right)$$

where f_{ij}^{in} and g_{ij}^{in} are the enumeration functions of the optimal matching of in-neighbors of nodes i and j with weight function $w\left(a, b\right) = x_{ab}^{k}$. Then we iterate and calculate the similarity of the each nodes until the similarity scores are getting converge.

The similarity of graphs G_A and G_B can be computed by

$$s\left(G_A, G_B\right) = \frac{1}{n}\sum_{l=1}^{n} x_{f(l)g(l)}$$

where f and g are the enumeration functions for the optimal matching and $n = min(\left|V_A\right|, \left|V_B\right|)$.

C. Feature Matrix

Only relying on the CFGs matching may lead the result of similarity detection unreliable, thus in this phrase we extract some features and define a feature matrix to enhance the comparison result obtained previously.

Let's assume there are two binary images A and B. A consists of m basic blocks, and A_i denotes one of them. Similarly, B consists of n basic blocks, and B_j denotes one of them. For any basic block A_i, whether there is an equivalent basic block B_j in B is unknown.

On the other side, let's assume we already know the CFGs of A and B are similar to a great degree after using our approximate graph matching technique. If these two binary images are exactly the same, they must share some equivalent basic blocks. Therefore, we can take advantage of this heuristic to improve the accuracy of similarity detection.

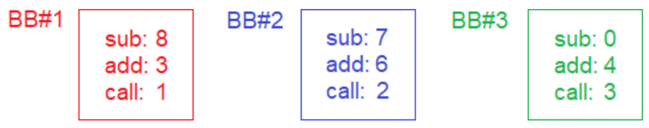


Figure 3: Three Basic Blocks

More specifically, for each basic block A_i, we enumerate the number of operators in it. For example, in Figure 3, there are three simplified basic blocks, the number of operators that appear in each basic block is counted and shown in each box.

Also, some small basic blocks which only contain tiny number of operators are very trivial. In order to reduce the computation complexity, it is reasonable to concentrate on such basic blocks which contain rich operators and ignore those small trivial counterparts. Based on this assumption, we design the feature matrix based on sorting.

Take the operators enumeration result of basic blocks shown in Figure 2 as an example. At first, we sort the basic blocks according to the number of operator 'sub' that they have. Then we obtain a sequence of basic blocks $\{A_1', A_2', A_3'\}$. Meanwhile we define a vector of operators {sub, add, call}, thus we have

- The number of operator 'sub' in A_1' is 8;
- The number of operator 'add' in A_2' is 6;
- The number of operator 'call' in A_3' is 3.

Then we use the first column of matrix shown in Table I to record the first feature vector {8, 6, 3}. Similarly, we sort the basic blocks according to the numbers of operator 'add' and 'call' that they have and finally generate the intact feature matrix as shown in Table I. Notice that Table I only presents a simplified feature matrix as an example. In practice, we will utilize a larger representative one.

Table I: An Example of Feature Matrix

	sub	add	call
sub	8	7	0
add	6	4	6
call	3	1	1

A pair of representative feature matrices of binary images A and B are obtained according to the approach discussed above. We use M_A and M_B to denote those two matrices. At last, the cosine similarity is used to measure the similarity between M_A and M_B. This metric will help us to confirm the similarity detection result from CFGs matching.

D. System Architecture

The overall scheme and pipeline are shown as Figure 4. First of all, we disassemble the binary images to obtain corresponding CFGs. Then Some approximate graph matching algorithms will be leveraged to find out the similarities among the CFGs. Meanwhile, we also can extract some features from the generated basic blocks and use the feature matrices to compare the released binaries with our reference binaries to detect the similarities. Finally, these two comparison results will cooperate together to generate a more reliable result.

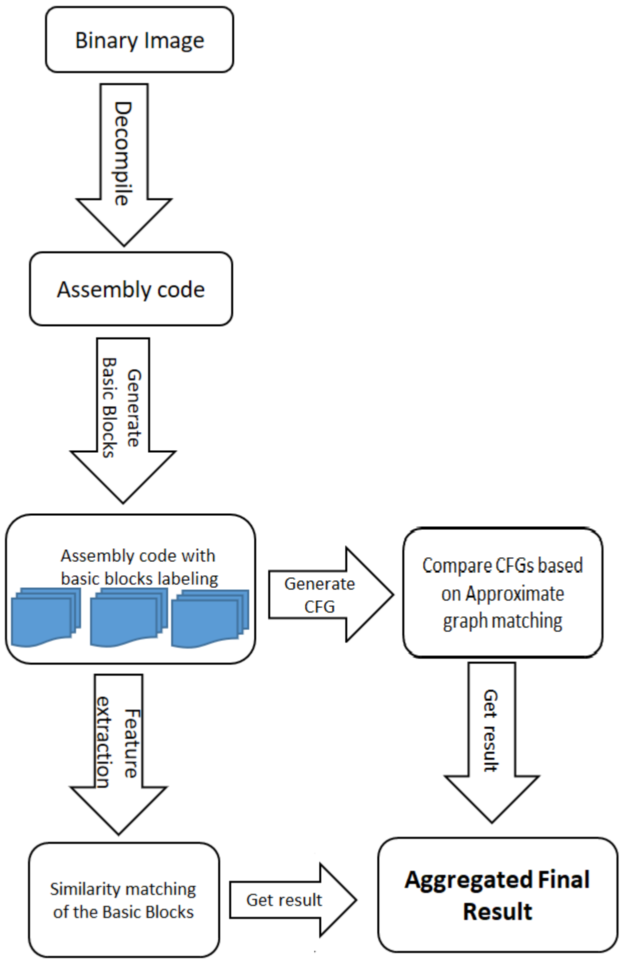


Figure 4: System Architecture

IV. Experiments and Evaluation

Overall, in this section we will evaluate the feasibility of our CFG generation method and graph matching mechanism.

To evaluate the proposed scheme, some practical binaries are used. For instance: OpenSSL and Linux coreutils. The former one is very popular and widely used in mobile operating systems and IoT devices. On the other hand, the latter one provides very common utility functionality in Linux-based IoT or mobile systems. In particular, to obtain two somewhat similar target binaries, we compiled the OpenSSL program with two different compilers, i.e., gcc and clang. After that, the CFGs from those two binaries (which are generated from two different compilers) are obtained. Observation indicates that the CFGs is not exactly same. The binary generated by gcc has 6847 nodes in its CFG, while the binary from clang contains 6839 nodes in its CFG. This difference implies that the different compilers perform distinct approaches or optimization to generate binaries from the same pieces of source code.

Moreover, we divide those binaries used in this experiment into three different groups: Small (the number of those nodes in CFG < 2500), Medium ($2500 \leq$ the number of those nodes in CFG < 5000) and Large ($5000 \leq$ the number of those nodes in CFG) considering the number of nodes in the CFG (See Table II).

Table II: Experimental Binaries

Binary Name	#Nodes in CFG	Group
ln	217	Small
kill	139	Small
tar	1329	Medium
OpenSSL (gcc)	6847	Large
OpenSSL (clang)	6839	Large

Then we carry out some comparisons between CFGs from different groups with the approximate matching approach mentioned in Section III.

Figure 5 demonstrates the outcome of our comparison. Generally speaking, the matching scores from our approach can reveal the similarity degree of two binaries. Even though the two versions of OpenSSL are compiled from different tools and the number of basic blocks in these two binaries are also distinct, our graph matching result still indicates they are very similar.

In addition, it can be observed that if the comparison occurs between binaries from different groups, then the similarity score is unusually high. The possible reason is that when we compare a small graph with a very big graph, the graph matching algorithm based on neighborhoods can find a similar sub-graph from the big graph. It is possible that such sub-graph will easily match the small graph. Based on this observation, we should not compare two graphs if the gap between those two numbers of their nodes is bigger than 25%. We can assume that if the difference of node counts is 25%, the binaries are totally different.

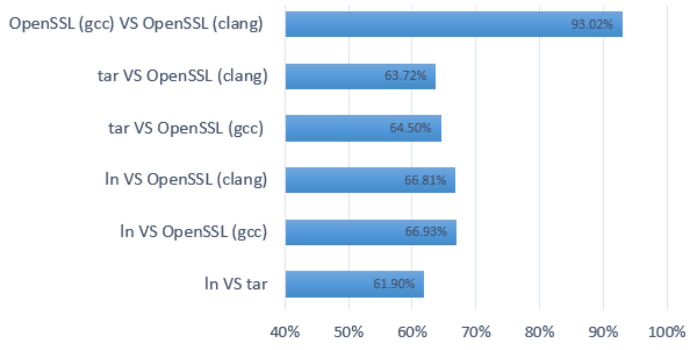


Figure 5: Similarity Scores of CFGs

Also, we observed that for small binaries the similarity score is much higher than the large binaries. For example: when we compare two versions of OpenSSL binaries, the result is 93.02%, but when we compare two different versions of ln, the result is 100%. Consequentially, in practice our advice is to set different thresholds for different groups of binaries to make the final judgment whether they are similar or not.

In order to make the similarity detection result from previous stages more reliable, as described before, we extract two representative feature matrices respectively from a pair of binaries. See Appendix for some examples of feature matrices, where the vector of operators we define is {sub, add, push, pop, call, cmp, lea}.

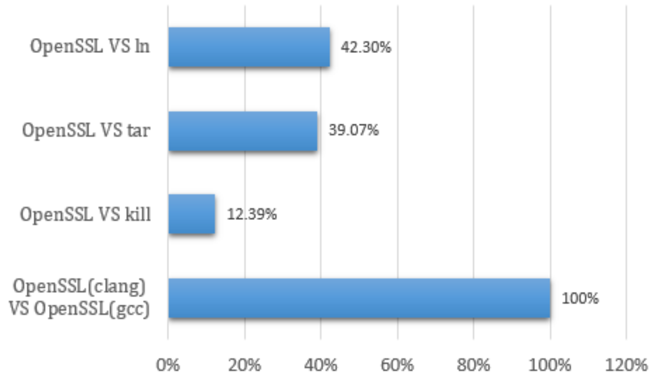


Figure 6: Cosine Similarity of Matrices Pairs

Then we compute the cosine similarity between those two feature matrices, some experiment results are shown in Figure 6. As expected, the cosine similarity score from those two representative feature matrices of different versions of OpenSSL is very high (100%). By contrast, if we compute the cosine similarity between two different binaries, then the score is very low (from 12.30% to 42.30%). It is obvious that our feature matrices can well represent the differences between two distinct binaries. Therefore, our scheme is feasible and applicable in similarity detection of binary images in practice.

V. Conclusion and Future Work

In this project, we propose a novel scheme which can be used to detect similarity between binary images in IoT devices. Our evaluation based on practical images also confirm that the overall scheme can be workable in practice.

Finally, we will leave the following ideas as our future directions:

- It is worth exploring further for a better aggregation mechanism of the results from two modules (basic blocks feature matching and CFGs similarity matching) to increase the precision of the existing system.
- The current evaluation is only based on limited binaries. The experiments based on more binaries to generate comprehensive results from current implementation will be more convincing. In addition, that will provide important data to tune the graph matching algorithm and to set precise threshold values for different groups of binaries when carrying out CFGs similarity matching.
- Our current implementation focuses on the similarity detection among the binary images of the IoT devices. However, it should be able to extended to detect the similarities in mobile applications with GUI. To consider these scenarios, the framework and GUI implementations need to be excluded from the comparisons. To tackle this problem, one database including some basic applications or frameworks will be maintained. For instance, we can collect some simple Android applications with only GUI elements without any program logic. Then the basic blocks of those applications can be used as references of the GUI related basic blocks. We can maintain those information in a ”GUI reference database”. During the extraction of the basic blocks of the targeted binary, the system will try to find every basic blocks in the database. Thus the system can detected the basic blocks responsible for GUI or framework implementation. GUI or framework

basic blocks will be excluded during CFGs comparison and feature extraction. We believe this approach will be applicable to detect similarity when facing the mobile applications with GUI or frameworks.

- The approximate graph matching algorithm in this project will work fine if two graphs are sparse. An obvious limitation of the method is that the computation time is linear with respect to the product of graph sizes due to the size of the similarity matrix. As for the future work, we are planning to look for or design an algorithm with lower time complexity that also works well with dense nodes.

REFERENCES

[1] M. M Mahmud Hossain, M. Fotouhi, R. Hasan, Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things, Services (SERVICES), 2015 IEEE World Congress.

[2] Z. Durumeric, J. Kasten, D. Adrian, J. A Halderman, M. Bailey, The Matter of Heartbleed, IMC '14 Proceedings of the 2014 Conference on Internet Measurement Conference

[3] F. Qian, Z. Rundong, X. Chengcheng, C. Yao, T. Brain, Y. Heng, Scalable Graph-based Bug Search for Firmware Images, CCS 16, October 24-28, 2016, Vienna, Austria

[4] T. Kamiya, S. Kusumoto and K. Inoue, 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7), pp.654-670.

[5] Z. Li, S. Lu, S. Myagmar and Y. Zhou, 2006. CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE Transactions on software Engineering, 32(3), pp.176-192.

[6] H. Sajnani, V. Saini, J. Svajlenko, C.K. Roy and C.V. Lopes, 2016, May. SourcererCC: Scaling code clone detection to big-code. In Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on (pp. 1157-1168). IEEE.

[7] S. Kim, S. Woo, H. Lee and H. Oh, 2017, March. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In Security and Privacy (SP), 2017 IEEE Symposium on. IEEE.

[8] J. Pewny, F. Schuster, L. Bernhard, T. Holz and C. Rossow, 2014, December. Leveraging semantic signatures for bug search in binary programs. In Proceedings of the 30th Annual Computer Security Applications Conference (pp. 406-415). ACM.

[9] M. Sergey and G.Molina, Hector and Rahm, Erhard , Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In 18th International Conference on Data Engineering (ICDE 2002), February 26 - March 1.

[10] D. C Schmidt, L. E Druffel , A Fast Backtracking Algorithm to Test Directed Graphs for Isomorphism Using Distance Matrices, Journal of the ACM (JACM),Volume 23 Issue 3, July 1976.

[11] L.P Cordella , P.Foggia, C. Sansone, F.Tortorella, M. Vento, Graph matching: A fast algorithm and its evaluation, ICPR '98 Proceedings of the 14th International Conference on Pattern Recognition-Volume 2 ,1998.

[12] Hex-Rays: The IDA Pro disassembler and debugger. www.hex-rays.com/idapro, 2011.

[13] G. Balakrishnan, R. Gruian, T. Reps, T. Teitelbaum, 2005, CodeSurfer/x86a platform for analyzing x86 executables. In: Proceedings of the 14th International Conference on Compiler Construction (CC). pp. 250-254.

[14] J. Kinder, H. Veith, 2008, Jakstab: A Static Analysis Platform for Binaries. In: Gupta A., Malik S. (eds) Computer Aided Verification. CAV 2008. Lecture Notes in Computer Science, vol 5123.

APPENDIX

The representative feature matrix of ln

$$\begin{bmatrix} 0 & 3 & 2 & 1 & 0 & 0 & 3 \\ 2 & 0 & 2 & 1 & 5 & 4 & 1 \\ 1 & 1 & 1 & 0 & 2 & 2 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 6 & 0 & 6 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The representative feature matrix of tar

$$\begin{bmatrix} 2 & 7 & 1 & 1 & 0 & 2 & 1 \\ 1 & 1 & 0 & 2 & 14 & 0 & 4 \\ 1 & 1 & 0 & 0 & 0 & 3 & 0 \\ 6 & 3 & 3 & 0 & 3 & 5 & 6 \\ 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 1 \\ 4 & 3 & 2 & 0 & 1 & 0 & 7 \end{bmatrix}$$

The representative feature matrix of kill

$$\begin{bmatrix} 0 & 3 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 23 & 46 & 3 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 3 & 2 \\ 0 & 0 & 0 & 3 & 2 & 0 & 0 \\ 6 & 0 & 2 & 0 & 3 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The representative feature matrix of OpenSSL

$$\begin{bmatrix} 158 & 390 & 12 & 2 & 0 & 0 & 342 \\ 0 & 0 & 1 & 17 & 96 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 5 & 0 \\ 31 & 0 & 1 & 30 & 0 & 0 & 64 \\ 5 & 0 & 0 & 6 & 1 & 0 & 0 \\ 0 & 0 & 19 & 0 & 0 & 0 & 0 \\ 16 & 1 & 1 & 0 & 0 & 0 & 2 \end{bmatrix}$$