Detection of Similarity in Cross Version Binaries using Raw Bytes

Nandish M.
Dept. of CSE, JNNCE, Shimoga

Mohan H.G.
Dept. of CSE, JNNCE, Shimoga

ABSTRACT
Binary code similarity detection (BCSD) technique compares multiple parts of binary code like functions, basic blocks or entire program to check for similarity or differences. Without relying on the source code, binary code analysis allows analysing code. BCSD is used for malware clustering, software theft detection and bug search. Existing techniques for BSCD problem includes Control Flow Graphs (CFG) and deep learning models. Here, a new and simple approach based on single feature to solve the cross-version BCSD problem is proposed. Approach follows initial transformation from functions to vectors and then computes the coefficient value. Proposed approach works on the raw bytes which is implemented and evaluated on a custom dataset having around 23,451 samples. The result shows that the model outperforms all other solutions and the recall of the approach could reach 97.1%.

Keywords
Cross Version Binary, Control Flow Graph, Similarity coefficient, Malware Detection

1. INTRODUCTION

Binary Code Similarity Detection (BCSD) is about deciding that any two binary blocks are identical, similar, and equivalent. The BCSD techniques are used for malware detection [1] and vulnerability search. The major issues in BCSD problems are binary codes can be generated using different compilers, architectures and codes of different versions. Cross-compiler binaries are generated when a source code is compiled with different compiler algorithms. The source codes compiled on different instruction set generates cross-platform binaries. The cross-version binaries are generated when the source codes are patched and evolved over time [2]. The cross-compiler and cross-platform binaries are said to be semantically equivalent if their functionality is same but having different syntactic features. The cross-version binaries are equivalent, since they are compiled in the same platforms having same root. The cross-version binaries still may have contrasting syntactic and semantic features. The available techniques for BCSD problems depend on binary functions graph-isomorphism and CFG methods [3].

Bindiff is predominantly used as a tool to compare binary functions CFGs and to identify the similarity among them [4]. BinHunt and iBinHunt uses symbolic execution and taint analysis to solve the BCSD problem [5]. In BinGo and Esh, the CFG are split into separate blocks to process them, which enhances the robustness of the model by evaluating the entire CFG and CFG fragments similarities [6]. The computation cost is reduced in Genius and Gemini method by extracting numeric features from basic blocks or CFGs. The neural network-based approach proposed by Gemini is used to compute the embedding, where embedding is a numeric vector derived from the CFGs of each binary blocks[7]. Here the similarity detection is done by evaluating the distance between the vectors of two blocks. This method shows very good results [8].

DiscovRE [9] method pre-filters the numerical features of CFGs and applies the KNN algorithm to find a list of candidate functions. This method extracts the syntax level features such as the number of arithmetic instructions/ call instructions, these are lightweight in nature which speed up the feature extraction process. Here the search efficiency is increased by applying graph matching after pre-filtering on function level features[10]. It is observed in Feng method [11] that, the pre-filtering approach to detect similarity rely on pairwise graph matching and is not highly reliable which degrades the search accuracy and becomes inefficient. The Genius approach [12] provides robustness against the code variations by using the Attributed CFGs (ACFGs) which are based on basic block features [13].

In this paper, Detection of Similarity in Cross Version Binaries are presented. The remaining sections of the paper is described as follows. The Section 2 gives an overview of Binary Code Similarity are described. In Section 3, the proposed method is explained along with an algorithm the experiment is detailed in Section 4. Conclusions of proposed method are drawn in Section 5.

2. VERSION BASED BCSD PROBLEM

The version based BCSD problem deals with the analysis of two binaries B1 and B2 which are compiled from a same project source code, that have evolved over time due to the need of the user. The cross-version BCSD has following computations:

- Function Matching: Finding the equivalent block for B1 in a function F1i for each function F2j having a binary B2.
- Similarity Score: Evaluating the similarity score for each pair of functions (F1i, F2j), the similarity score ranges between (0, 1). This score indicates the equivalency between the two binaries.
- Difference Identification: for each pair of functions (F1i, F2j), If the Similarity Score < 1 then the differences can be identified.

3. PROPOSED SYSTEM

The architecture of the proposed method is depicted in the Figure 1. The binary functions are fed as input to the model. The binary functions are preprocessed to extract the byte values. These byte values are represented as numerical vectors. The two numerical vectors are processed to identify the similarity using a matching function described in section 2. The similarity coefficient of the matching function is used to detect the degree of similarity of the two input functions.

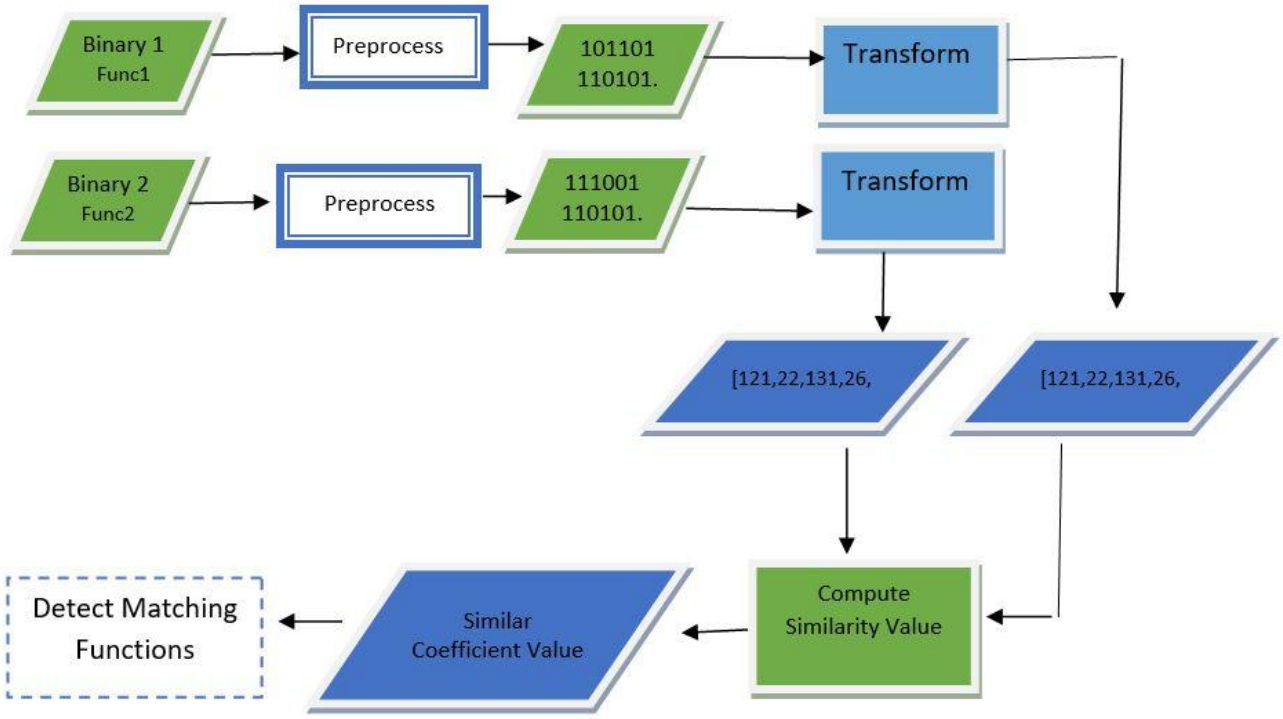

Fig 1: Architecture of proposed BCSD method

The algorithm for the proposed BCSD method is as follows:

1. **Input**: Two Binary Codes say Z1 and Z2
2. Extract a Binary Function say F1i from the Z1
3. Extract a set of Binary Functions called Processing List {F21, F22, …, F2j, …, F2n} from the Z2
4. Set the length of F1i as value for len i.e., size of the binary block
5. **foreach** Binary Function in {F21, F22, …, F2j, …, F2n}
   if length(F2j) < len **then**
   Remove F2j from the processing list
6. **foreach** F2j in Processing List
   for x from 1 to len
   Decompose F1i into x number of parts say P1, P2,…, Px
   foreach Pk in { P1, P2,…, Pi }
   matchCount[] = compute SimilarityCoefficient(Pk, F2j)
   SCij[] = sumofall(matchCount) / x
7. Compute simT = $\sum_{q=0 \text{ to len}}$ SCij[q] / q, where q is the index of array SCij
8. Similarity value (F1i, F2j) = sigmoid(simT)
9. **if** Similarity value (F1i, F2j) > 0.5 **then**
   Two Binary Functions F1i and F2j are similar
10. **else** Binary Functions F1i and F2j are different.

The proposed BCSD method compares two binary functions to check for similarity. The method follows the sliding window approach while comparing two series of binary byte values. The method considers only those functions having their byte length greater than the first function (F1i). Initially partition size is set to one. Each partition of F1i is compared with the F2j, here the comparison will be done at the byte level. If the byte sequence are same then match count is increased. Later the comparison window is moved by one byte position for further comparison till one reach the end of F2j. The process is repeated on subsequent partitions made on F1i until one reach a stage where length of binary code on partitioning is one byte.

The similarity between F1i and F2j is evaluated based on a summation of scores obtained on each partition as described in the Algorithm 1 and 2. The weighted sum is used to compute the Similarity value so that the largest binary code sized partition will have more weightage. The Binary functions F1i and F2j are said to be similar if the Similarity value is more than 0.5. The Similarity value is calculated by applying a sigmoid function to normalize the values in the range of (0,1).

The algorithm for finding Similarity Coefficient

1. SimilarityCoefficient(Pk, F2j)
2. MatchCount = 0
3. **foreach** byte pos from 0 till bytelength(F2j) – bytelength(Pk)
4. Compare byte values of Pk and F2j at index pos
5. **if** byte values of Pk == F2j **then**
   a. MatchCount = MatchCount + 1
   b. **return** MatchCount

4. EXPERIMENT

4.1 Implementation and Setup

The proposed BCSD method is implemented in Python Language on Intel i7 machine with 2.2GHz CPU having 8GB of RAM. The method is run on Alpha-Diff Dataset. In this dataset the binary functions can be clearly identified. There are 2,489,793 positive samples (matching functions) in the dataset from 66,823 pairs of cross-version binaries. These binaries are from x86 Linux platform. For experimental test data set, method used a batch command for random selection and manual sample selection was avoided. 343,178 functions are used in test data for cross-version matching, from 11,000 pairs of different-version binaries. The test dataset is divided into small and big subsets.

4.2 Accuracy and Efficiency of Testing Stage

In testing stage of proposed method, 11,000 pairs of different version binaries which includes 343,178 positive samples are used. The method is evaluated on the entire test dataset. The Recall Rate is used as a metric to evaluate the performance of the system. The Recall results are shown in Table 1. It is evident from the experimental results that the proposed BCSD method is successful in evaluating the similarity in cross-version binary code.

From Fig.2, one can note that the test time is in standard increase with increase in binary size and function size. The method based on the coefficient of similarity of Jaccard stays the most efficient. In the database, the length of the most binary is shorter than 20,000 bytes and most functions are shorter than 1,000 bytes, which means to test, the method can get the same job within 4 seconds of multiple samples. In the study at a large subset, after we have set the length of the function, many jobs that do not meet the requirements are not included. I the length of the slide window usually does not do that exceed 100,000 bytes and the required time for running is about few seconds. But in practice, if sliding window check requires a large file size (alternative than 1MB), the operating time of the algorithm is about tens of seconds, which is not appropriate. However, we can split the large file into many smaller files and use a distributed algorithm to improve efficiency in future work.

Table 1. The recall accuracy on Alpha-Diff Dataset

	Small Dataset	Big Dataset
Recall @1	0.9552	0.9011
Recall @2	0.9722	0.9422

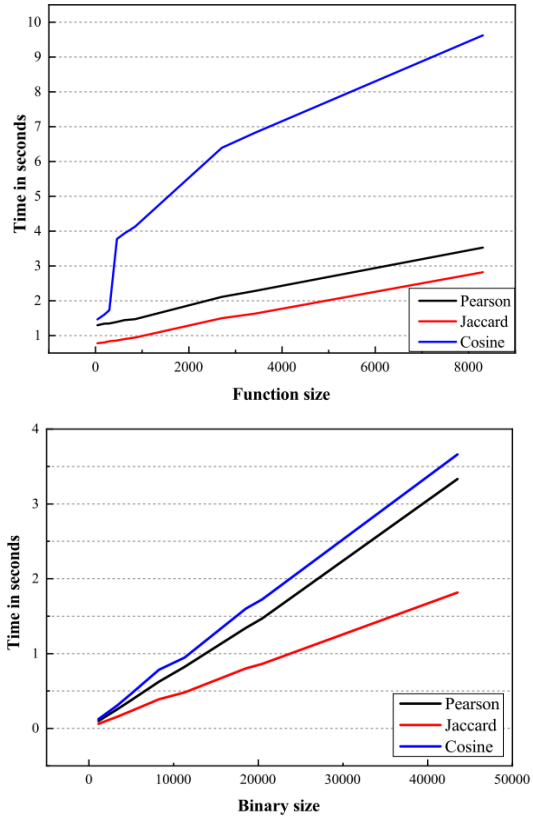

Fig 2: Efficiency evaluation on Alpha-Diff dataset

5. CONCLUSION

The approach for BCSD mentioned in this paper identifies the similarity in two binary functions generated by cross-version source code. The method can be implemented efficiently since it do not involve any complex algorithmic computations unlike other methods mentioned in section 1. The method is applied on raw bytes directly without using any complex graphical methods or Deep Learning methods. The more vulnerable firmware images can be identified using this method. The method can be extended to identify the cross-architecture and cross-compiler binaries. The area of BCSD is still in evolving to identify the semantic similarity purely based on the binary code irrespective of how they are generated.

6. REFERENCES

[1] Hui Guo , Shuguang Huang, Cheng Huang, Min Zhang, Zulie Pan, Fan Shi, Hui Huang1, Donghui Hu And Xiaoping Wang , "Lightweight Cross-Version BCSD Based on Similarity and Correlation Coefficient Features", IEEE Access, pp. 120501 - 120512, Vol 8, June 2020.

[2] X. Hu, T.-C. Chiueh, and K. G. Shin, ''Large-scale malware indexing using function-call graphs'', in Proc. 16th ACM Conf. Comput. Commun. Secur., 2009, pp. 611–620.

[3] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary" in Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE). New York, NY, USA: Association Computing Machinery, 2018, pp. 896–899.

[4] Y. David, N. Partush, and E. Yahav, ''Statistical similarity of binaries,'' ACM SIGPLAN Notices, vol. 51, no. 6, pp. 266–280, Aug. 2016.

[5] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Crossarchitecture bug search in binary executables", in Proc. IEEE Symp. Secur. Privacy, May 2015, pp. 709–724.

[6] L. Massarelli, G. A. D. Luna, F. Petroni, L. Querzoni, and R. Baldoni, "SAFE: Self-attentive function embeddings for binary similarity", in Proc. 16th Conf. Detection Intrusions Malware Vulnerability Assessment (DIMVA), 2019, pp. 309–329

[7] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "BinGo: Cross-architecture cross-OS binary search", in Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE), 2016, pp. 678–689.

[8] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs", in Proc. Netw. Distrib. Syst. Secur. Symp., 2019, pp. 1–15.

[9] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2 Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization", in Proc. IEEE Symp. Secur. Privacy (SP), May 2019, pp. 472–489.

[10] J. Ming, M. Pan, and D. Gao, "IBinHunt: Binary hunting with interprocedural control flow", in Proc. Int. Conf. Inf. Secur. Cryptol. Berlin, Germany: Springer, 2012, pp. 92–109.

[11] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs", in Proc. Int. Conf. Inf. Commun. Secur. Berlin, Germany: Springer, 2008, pp. 238–255.

[12] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li,Feng Li,Aihua Piao,Wei Zou, "αDiff: Cross-Version Binary Code Similarity Detection with DNN" , ACM, pp. 667- 678 , September 3–7, 2018.

[13] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "DiscovRE: Efficient cross-architecture identification of bugs in binary code", in Proc. Netw. Distrib. Syst. Secur. Symp., 2016, pp. 1–15.