IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 30, 2020, accepted June 20, 2020, date of publication June 25, 2020, date of current version July 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3004813

A Lightweight Cross-Version Binary Code Similarity Detection Based on Similarity and Correlation Coefficient Features

HUI GUO[1], SHUGUANG HUANG[1], CHENG HUANG[2], (Member, IEEE), MIN ZHANG[1], ZULIE PAN[1], FAN SHI[1], HUI HUANG[1], DONGHUI HU[3], (Member, IEEE), AND XIAOPING WANG[1]

[1]College of Electronics Engineering, National University of Defense Technology, Hefei 230011, China
[2]College of Cybersecurity, Sichuan University, Chengdu 610065, China
[3]School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230009, China

Corresponding author: Min Zhang (dyzhangmin@163.com)

This work was supported in part by the Laboratory of Network Security, College of Electronics Engineering, National University of Defense Technology, and in part by the Provincial Natural Science Foundation of Anhui under Grant 1908085QF291.

ABSTRACT The technique of binary code similarity detection (BCSD) has been applied in many fields, such as malware detection, plagiarism detection and vulnerability search, etc. Existing solutions for the BCSD problem usually compare specific features between binaries based on the control flow graphs of functions from binaries or compute the embedding vector of binary functions and solve the problem based on deep learning algorithms. In this paper, from another research perspective, we propose a new and lightweight method to solve *cross-version* BCSD problem based on multiple features. It transforms binary functions into vectors and signals and computes the similarity coefficient value and correlation coefficient value for solving *cross-version* BCSD problem. Without relying on the CFG of functions, deep learning algorithms and other related attributes, our method works directly on the raw bytes of each binary and it can be used as an alternative method to coping with various complex situations that exist in the real-world environment. We implement the method and evaluate it on a custom dataset with about 423,282 samples. The result shows that the method could perform well in *cross-version* BCSD field, and the recall of our method could reach 96.63%, which is almost the same as the state-of-the-art static solution.

INDEX TERMS Binary code similarity detection, cross-version binary, malware detection, similarity coefficient, correlation coefficient.

I. INTRODUCTION

Evaluating whether two binary functions are similar or not is known as binary code similarity detection (BCSD), which has been applied in many fields, such as malware detection [1], [2], malware family analysis [3] and plagiarism detection [4], [5]. The technology of BCSD could also be used to analyze vulnerabilities [6], [7] and search bugs [8]–[10], when applying it on known vulnerabilities and target applications. However, the problem of BCSD faces many challenges, such as cross-compiler BCSD problem, cross-architecture BCSD problem and cross-version BCSD problem. First, the source code compiled with different algorithms yields cross-compiler binaries. Then, the source code compiled on different platforms generates cross-architecture binaries. Last, the source code may be patched and evolve over time, generating cross-version binaries. The cross-compiler and cross-architecture binaries are semantic-equivalent, but they have different syntactic structures, whereas the cross-version binaries are similar to each other by nature, because they are compiled in the same environment, and they have a same root. But the cross-version binaries still have different syntactic structures and slightly different semantic features.

Most existing solutions for BCSD problems rely on the control flow graphs (CFGs) of functions in binary and graph-isomorphism algorithms. The widely used tool Bindiff [11] compares functions' CFGs and their attributes

The associate editor coordinating the review of this manuscript and approving it for publication was Fan Zhang.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

to solve the BCSD problem. BinHunt [12] and iBinHunt [13] utilize taint analysis and symbolic execution to handle these challenges. CABS [14], BinGo [15] and Esh [16] divide the CFG into different parts, and improve their robustness to CFG changes by computing the overall CFG and CFG fragments similarities. To minimize the cost of the computation, DiscovRE [8], Genius [9], Gemini [10] and VulSeeker [17] extract some numeric features from CFGs or basic blocks. DiscovRE [8] employs these features as a pre-filter on CFGs and uses the KNN algorithm to identify a small set of candidate functions. Genius [9] uses the basic block features to obtain the Attributed CFGs (ACFGs) and provides more robustness to code variation. Gemini [10] uses a neural network to compute the embedding of ACFGs and get better performance. VulSeeker [17] uses the CFG and DFG (data flow graph) to construct the labeled semantic flow graph (LSFG) and captures more function semantics and acquires a higher accuracy and efficiency.

A. NEED FOR THIS STUDY

These solutions rely on the CFG of functions, which was derived from expertise to construct semantic features of binaries. These methods have achieved good research results in multiple code similarity detection fields and have many important applications. However, sometimes the CFG of binaries could change dramatically while there may be only a little change in binary codes. And in some special cases, when we cannot extract effective CFGs and their related features, or the CFG features of the binaries are intentionally modified, how can we solve cross-version BCSD problems? So the first problem addressed in this paper is:

P1: *How to solve cross-version BCSD problems without relying on the CFGs and their related attributes?*

Asm2Vec [18], INNEREYE [19] and SAFE [20] explore many new methods to compute the embedding vector of binary functions. Asm2Vec [18] employs representation learning to construct a feature vector for assembly code and provides more robustness to code obfuscation and compiler optimizations. INNEREYE [19] utilizes word embedding and LSTM to automatically capture the semantics and dependencies of instructions and solves the BCSD problem among basic blocks. SAFE [20] proposes a new architecture for computing binary function embeddings from disassembled binaries and get better performance. Besides, Alpha-Diff [21], which is one of the state-of-the-art solutions to solve cross-version BCSD problems, represents the raw bytes of functions as images and uses the Siamese convolutional neural network to compute the similarity of functions.

Solutions based on deep learning and natural language processing algorithms are the current research hotspots in the BCSD field. They can automatically learn what is important from the assembly code and raw bytes, and provide better performance. But the deep learning algorithms still have some limitations, such as the problem of deep learning adversarial examples. In the field of malware detection and plagiarism detection, hackers may intentionally modify samples to evade the detection of deep learning models. Thus, in some special cases, we still need other methods to solve cross-version BCSD problems. Moreover, a new and different method can enable users to better cope with the complex real-world environment. So the second problem addressed in this paper is:

P2: *How to solve cross-version BCSD problems without deep learning algorithms?*

Binary Code similarity detection technology is currently mainly used in the fields of vulnerability mining, malware detection, and plagiarism detection. Among them, in the field of vulnerability mining, researchers mostly work on high-performance servers or hosts, and lightweight devices are generally not used; In the field of malware variant detection, high-performance servers can achieve better detection results. However, when the new malware variant is transmitted to the terminal device (i.e. PC and IoT devices), if the suspicious code can be first screened by the detection technology deployed on the terminal device before uploaded to the cloud, we can save some resources and improve efficiency. It is of significance to solve BCSD problems with these lightweight devices.

The main features that the above solutions rely on are CFG of function, assembly code and raw bytes. However, in some lightweight devices, the CFG of function and assembly code may not be easily extracted. Therefore, the solutions based on raw bytes are more suitable for solving problems in this field. There are many pure syntactic solutions based on raw bytes, such as similarity detection based on opcodes. And they can be performed by Yara search or other tools. But most of them need to manually analyze some unique attributes of the application target, such as some signatures or other characteristics, and the application is not for all cross-version binaries. So the third problem addressed in this paper is:

P3: *How to extract valuable features from the raw bytes of functions?*

B. MAJOR CONTRIBUTIONS OF THE STUDY

Aiming at these problems, we propose a method for cross-version binary code similarity detection. In short, it extracts some proper features from binaries and uses them to detect the similarity of binaries. In the method, each function in binaries would be transformed into vectors and signals. It would compute the similarity coefficient value and the correlation coefficient value of them.

First, we do not use the CFG of functions and related features but focus on the raw bytes of binaries to extract the related features. Second, each byte of the binary could be converted into a vector. And we also represent the raw bytes as signals. Third, instead of using deep learning methods and convolutional neural networks, our method detects the similarity of binaries based on the similarity coefficient methods and correlation coefficient methods. These two types of methods are relatively easy to deploy on lightweight

devices and do not need to collect numerous samples for training.

Thus, given a pair of cross-version binaries, we could represent them as vectors and signals and compute the similarity coefficient values and correlation coefficient values. Finally, we would take these values into account to solve the cross-version BCSD problem.

We have evaluated the method on a custom dataset consisting of 423,282 pairs of cross-version functions, which are collected from GitHub, Debian and other public repositories. The result shows that the recall of the method could still reach 96.63%. It proves that from another research perspective, our method could perform well in the field of cross-version binary code similarity detection without relying on CFG, deep learning algorithm and other related attributes. Together with various solutions such as Bindiff, Gemini, Alpha-diff, etc., our method can be used as an alternative solution to cope with various complex situations that exist in the real-world environment.

Overall, we make the following contributions:

1) Without relying on the CFG of functions, deep learning algorithms and other related attributes, we propose a new and light-weight method to extract features from the raw bytes of functions and solve cross-version BCSD problems, which could solve some limitations that may exist in the existing research.
2) Our method extracts the function features from the raw bytes of binaries. Together with the signal processing technique, the method could perform well on cross-version BCSD problems. The method would be proof that the signal processing technique is a viable approach to solving the BCSD problem.
3) We evaluate the method on a custom dataset, which consists of 12,000 pairs of binaries and 423,282 pairs of functions. The result shows that the method could perform well on the dataset, and the recall of the method could reach 96.63%.

The rest of the paper is organized as follows. In Section II, we introduce some definitions related to the BCSD problems. In Section III, we show how to represent the functions as signal waves and detect the binary code similarity. The experiment is detailed in Section IV. Section V provides some discussion related to our method. Section VI describes the limitation and future work of the study. In Section VII, we introduce the related works about the solutions for BCSD problems. Section VIII provides the conclusion of the study.

II. PROBLEM DEFINITION

In this section, we will introduce some definitions related to the cross-version binary code similarity detection (BCSD) problem.

A. NOTATIONS AND ASSUMPTIONS

A binary X_i includes a set of functions $f_{i1}, f_{i2}, \ldots, f_{in}$. We assume each function f_i can be identified correctly by existing technology solutions. And to be practical, we introduce the following three assumptions:

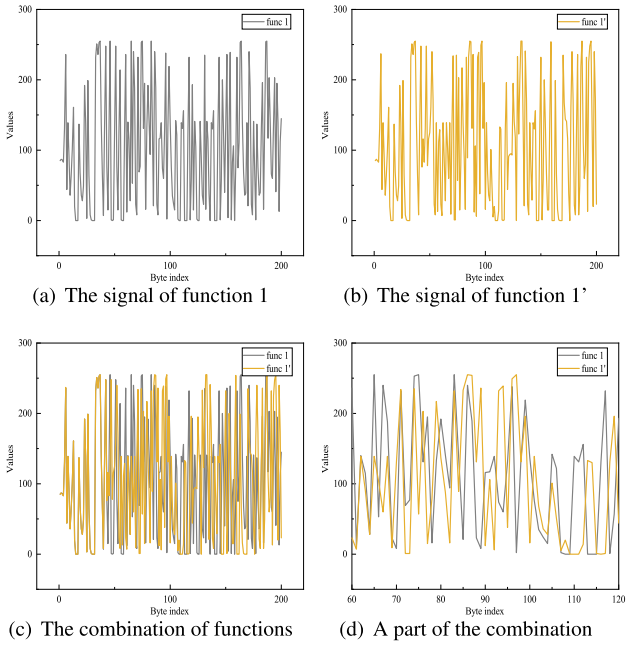
1) All the binaries we used in the research are compiled in high-level languages;
2) All the binaries are not generated with any obfuscated skills (e.g. packer);
3) The debug symbols in binaries are stripped.

A core task of binary code similarity detection is to find the matching functions in the target binary based on the giving functions. Two functions f_i and f_j are considered as matching if their source codes are identical or have the same name, same namespace, same class, and they are used in similar contexts (i.e. same functions from a same source code project in different versions).

B. CROSS-VERSION BCSD PROBLEM

Cross-version binary code similarity detection is related to three problems:

1) *Function matching problem*: for each function f_{1i} in binary X_1, find its matching function f_{2j} in the binary X_2.
2) *Similarity score computing problem*: for each pair of matching functions, compute the similarity degree between them and give a score ranging from 0 to 1.



(a) The signal of function 1 (b) The signal of function 1'

(c) The combination of functions (d) A part of the combination

FIGURE 1. An example of a pair of functions in cross-version binary code. Fig 1(a) and Fig 1(b) show the signals of a pair of cross-version functions. Fig 1(c) shows the combination of the cross-version functions and Fig 1(d) shows a part of the combination.

C. CROSS-VERSION BINARY SAMPLE

Fig.1 shows an example of a pair of matching functions from cross-version binaries, which is included in the Alpha-Diff [21]. The Fig.1(a) and Fig.1(b) show the signals of the matching functions. The Fig.1(c) shows the

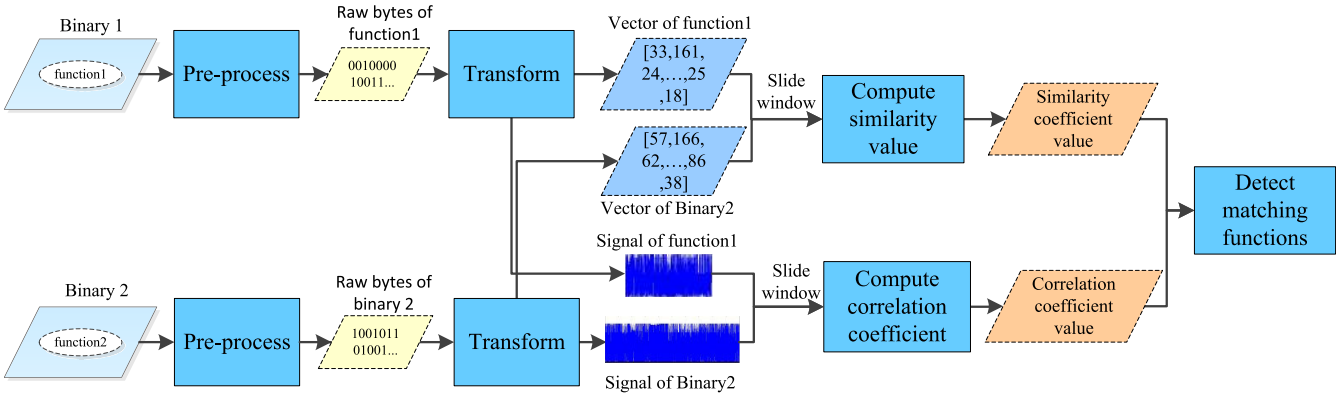


FIGURE 2. Overview of the method for cross-version binary code similarity detection.

combination of these two functions. We can see that the signals of the two functions have some similarity to each other, and there may be some connections between the two signals. Especially from Fig.1(d), we can see that although the values in signals are different, the trend (upward and downward) of the signals are similar with each other. And it may be another feature between cross-version functions.

III. APPROACH

A. OVERVIEW

As shown in Fig.2, our solution first extracts each byte value from the binary functions, and represents the functions as vectors and signals respectively, then computes the similarity coefficient value of vectors and the correlation coefficient value of the signals, and finally detects the matching functions based on these two values.

Unlike the Alpha-Diff solution which applies CNN to solve the cross-version BCSD problem, we use the vector's similarity methods to directly extract features from function's raw bytes and generate the similarity coefficient value to characterize the intra-function's semantic feature. Moreover, we use the correlation coefficient method to extract the function's linear correlation and characterize the function's linear semantic feature.

B. TRANSFORMATION

A given binary which includes functions can be represented as a string of zeros and ones. It could be read as a vector of 8-bit unsigned integers and reshaped into an array. This array can be viewed as a signal value in the range [0, 255]. Each function in the binary could also be represented as a vector and transformed into a signal. The length of vectors and signals depends on the size of binaries. The transformation process is shown in Fig.3.

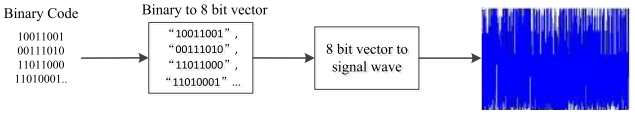


FIGURE 3. Represent binary code as vectors and signal waves.

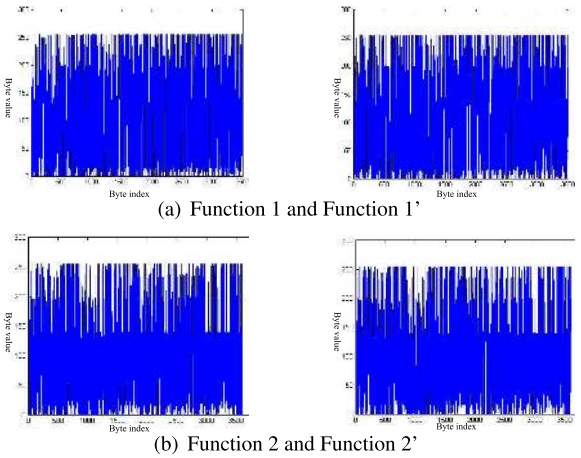


FIGURE 4. Byte plots of the different functions in cross-version binary code. Fig. 4(a) shows a pair of matching functions. Fig. 4(b) shows another pair of matching functions.

Fig.4 shows examples of two pairs of different functions from cross-version binaries, which are included in the dataset [21]. One notable observation is that the signals of the matching functions appear visually similar to each other while the others are distinct. The similarity of the signal also motivates us to find the matching functions by using the techniques from signal processing field.

C. MATCHING FUNCTION DETECTION

Algorithm 1 shows our process of solving the BCSD problem. As input, the algorithm takes a function f_{1i} in binary X_1, a binary X_2 including all the functions $f_{21}, f_{22}, \ldots, f_{2n}$, a minimum length γ of the function parts and the number D of the function partition. If exists, it returns the matching function f_{2j} in binary X_2 and proceeds in three basic steps: (1) generate the function candidate set X_2' based on the length of the function f_{1i} and divide the raw bytes of the function f_{1i} into D parts, (2) construct the similarity wave array S based on f_{1i} function parts and binary X_2', and (3) find the matching function f_{2j} based on the similarity wave array S. The details of three steps are as follows.

1) GENERATING FUNCTION CANDIDATE AND DIVIDING INPUT FUNCTIONS

The matching functions are compiled from functions with the same namespace, class and used in a similar context. And even the functions have the same raw bytes. Inspired by these characteristics of the matching functions in cross-version binaries, we can assume that the matching functions possess several similar attributes, Among which the length of a function is rather intuitive and important.

So we add a length limit to the functions in X_2 based on the function f_{1i}. The first step of the process is to generate the function candidate code X_2'. We set a length limit $[l_{1i}, l'_{1i}]$ based on the length l of the function f_{1i}, e.g. $[0.5l,\ 2l]$. For each function in binary X_2, if the length of functions adherence to the length limit, we add the function into function candidate set. And we use this function set to generate function candidate code X_2', as this allows us to find the matching function more precisely and quickly.

After generating function candidate code X_2', we would evenly divide the function f_{1i} into D parts. The initial value of D is one, which means at the beginning of the process, we compute the similarity of binary codes based on overall raw bytes of function f_{1i}. If the change is relatively noticeable in the functions, causing the similarity of overall matching functions not higher than the others, we would divide the functions into several parts (i.e. we would increase the value of D). Some function parts mainly consist of the unchanged parts of functions. The similarity of these parts would be more apparent than that of the others. As a consequence, we divide the function f_{1i} into several parts to detect the similarity of the binary codes.

Algorithm 1 Similar Functions Detection

f_{1i} is a function in binary X_1, X_2 is the binary code including all the functions $f_{21}, f_{22}, \ldots, f_{2n}$, D is the number of the function partition.

Input: f_{1i}, X_2, D
1: $x^* \leftarrow 0, X_2' = \varnothing, l = \dfrac{len(f_{1i})}{D}$
2: Generate the length limit$[l_{1i}, l'_{1i}]$ based on l
3: **for** each function f_{2j} **do**
4: **if** $len(f_{2j}) \in [l_{1i}, l'_{1i}]$ **then**
5: $X_2' = [X_2', f_{2j}]$
6: **end if**
7: **end for**
8: Divide f_{1i} into D parts
9: **for** each part f_{1i_m} **do**
10: $S[m] = Similarity\_wave(f_{1i\_m}, X_2')$
11: **end for**
12: **if** $x^* == 0$ **then**
13: get max$(S[m][x])$ and $x^* \leftarrow x$
14: **end if**
15: find f_{2j} in X_2' by the byte index x^*
16: **return** f_{2j}

2) GENERATING THE SIMILARITY WAVE ARRAY

The similarity wave array indicates which parts of the two binaries have high similarity and are therefore a visualization tool that allows us to find the match f_{2j} in binary X_2. The similarity wave array is calculated based on similarity coefficients method (e.g. Jaccard coefficient), as this gives users the information needed to find the matching function. More precisely, the user wants to find the matching function f_{2j} in binary X_2 based on the function f_{1i}. To this end, the similarity value of the matching function f_{2j} in binary X_2 given by the similarity wave array should be high while the similarity of other parts should be low. The user can accomplish this by using similarity wave array S. The process for the similarity wave array generation is given in algorithm 2.

Algorithm 2 Similarity Waves Generation

f_{1i_m} is a function's part of function f_{1i} in binary X_1, and X_2' is the binary code including the matching function.

Input: f_{1i_m}, X_2'
1: $l \leftarrow len(f_{1i\_m})$
2: Generate candidate set B in X_2' based on f_{1i_m} using slide window method
3: **for** each part $b \in B$ **do**
4: Compute similarity coefficient $S(f_{1i\_m}, b)$
5: $S[j] = S(f_{1i\_m}, b)$ j is the byte index of b in X_2'
6: **end for**
7: **return** S

As input, algorithm 2 takes a part f_{1i_m} of the function f_{1i}, the binary X_2 and proceeds in the following steps. (1) From the first byte to the end of X_2', generate several candidate parts in X_2' based on slide window method (the window size is equal to the length of the part f_{1i_m} and the slide size is equal to one byte). These parts constitute the set B according to their first-byte index in X_2'. A binary code usually includes a set of functions and the matching function is only a part of the binary. Based on the slide window method, we can find the matching function in binary code. Fig.5 shows the process of the slide window method.

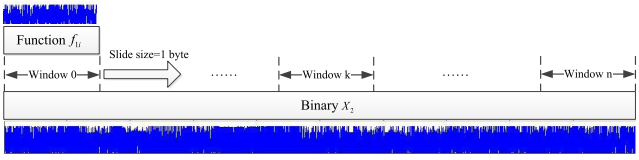


FIGURE 5. The implementation process of detecting the matching function based on slide window method.

As shown in Fig.5, the window size is equal to the length of the input functions in binary X_1, and the slide size is equal to one byte. From the first byte to the end of binary X_2, we generate several parts which are shown as the number of the windows from 0 to n in Fig.5. And we take each part b in binary X_2 to compute the similarity value with the input

functions. By using this method, any byte of X_2 would not be left out. If the binary X_2 includes a matching part with the input functions, we could find it. Moreover, we could also get the position of the matching part in X_2, which is of great significance in some applications.

In the next step, we would compute the similarity coefficient values based on each part $b \in B$ and f_{1i_m}. In the method, we firstly represent the raw bytes of functions as vectors and signals, then takes each part b to compute the similarity coefficient value with the part f_{1i_m} and uses these values to constitute similarity wave S by the position index of b's first byte in X_2'.

a: VECTORS

The similarity coefficient method is used to compute the distance and similarity value between the vectors. The vectors with the smallest distance would be the matching function's vector. Thus, we compute the similarity coefficient value to find the matching function. Because of the characteristics of the cross-version binaries, we select two similarity coefficient methods (i.e. cosine similarity, Jaccard similarity coefficient). Cosine similarity is defined as:

$$\mathrm{Cos}\,(\mathrm{A},\mathrm{B}) = \frac{\sum_{i=1}^{n}(A_i \times B_i)}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}} \tag{1}$$

where A_i and B_i represent the components of the vector A and B. And n isthe number of the components of A and B. Cosine similarity measures the similarity between two vectors by measuring the cosine of the angle between them and the values range from -1 to 1. When the value is a positive number, it indicates that the two variables are similar to each other. Jaccard similarity coefficient J is defined as:

$$\mathrm{J}\,(\mathrm{A},\mathrm{B}) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{\mathrm{A}+\mathrm{B}-|A \cap B|} \tag{2}$$

Jaccard similarity coefficient also called the Jaccard index, is used to compare the similarity and difference between finite sample sets. It is defined as the ratio of the intersection size of A and B to the union size and the values range from 0 to 1. When the value is equal to 1, it indicates A and B are the same set. When the value is equal to 0, it indicates A and B are different.

b: SIGNALS

We represent each byte as the instantaneous value of a signal and connect these instantaneous values to transform binaries into a discrete signal. We compute the correlation coefficient value of signal waves to find the matching functions. In the signal processing field, cross-correlation is used to measure the similarity of two signal waves between two different times in the time domain. The signal wave can be either a continuous signal or a discrete signal. The cross-correlation function is defined as:

$$\mathrm{R}\,(s,t) = E(X\,(s) * Y(t)), \tag{3}$$

where s and t are two different times and the $R(s,t)$ measure the degree of correlation of two signal $X(t)$ and $Y(t)$ between any two different times s and t. For two signal waves, the cross-correlation function is usually the complex function of time and is defined as:

$$\mathrm{R}(\,(X * Y)(\tau)) = \int_{-\infty}^{+\infty} X\,(t) * Y\,(t+\tau)dt, \tag{4}$$

where τ is a time moment, but the function's signal wave is a discrete signal, the cross-correlation function of discrete signal wave is usually defined as:

$$\mathrm{R}(\,(\mathrm{X} * \mathrm{Y})(\mathrm{n})) = \sum_{-\infty}^{+\infty} \mathrm{X}\,(\mathrm{m}) * \mathrm{Y}\,(\mathrm{m}+\mathrm{n}), \tag{5}$$

where n is a time moment and m changes from negative infinity to positive infinity. Pearson Correlation Coefficient is one of the most popular cross-correlation methods and it could measure the linear relation between two signal waves, which is suitable for solving cross-version BCSD problem. The Pearson Correlation Coefficient P is defined as:

$$\mathrm{P}\,(\mathrm{A},\mathrm{B}) = \frac{\mathrm{cov}(\mathrm{A},\mathrm{B})}{\sigma_\mathrm{A}\sigma_\mathrm{B}} = \frac{\mathrm{E}((\mathrm{A}-\mu_\mathrm{A})(\mathrm{B}-\mu_\mathrm{B}))}{\sigma_\mathrm{A}\sigma_\mathrm{B}} \tag{6}$$

where σ_A and σ_B are the standard deviations of A and B respectively. $cov(A,B)$ is the covariance of A and B. μ_A and μ_B are the means of A and B respectively. Pearson Correlation Coefficient describes the degree of linear correlation between two variables and the values range from -1 to 1. When the value is a positive number, it indicates that the two variables are positively linear correlated.

3) FINDING THE MATCHING FUNCTION

If the binary code X_2 includes the matching function of f_{1i}, there should be a part $b \in X_2'$ that would have a high similarity coefficient value with the part f_{1i_m}, which means the similarity wave S would include a fairly high value. In summary, the high value of $S[i]$ corresponds to the function that is the matching function f_{2j} in X_2'. By using the similarity wave S, we eventually find the matching function f_{2j} in X_2'. Similarity wave array samples are shown in Fig.6.

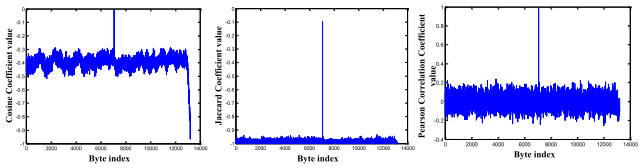


FIGURE 6. Similarity wave array samples.

Once a similarity wave array has been identified, it needs to be used to help the users to find the matching function. This is the last step in each iteration. If users get a wave array which is similar to the one shown as Fig.6, in which the value of the matching function part is higher than other functions, users could easily find the matching function in X_2. Nevertheless, if there does not exist a high value in similarity wave array, it means we can not get the matching function in X_2 for input part f_{1i_m}, for the reason that the functions or parts

have changed a lot, which decreases the similarity coefficient value of them. However, the smaller part of the function may mainly consist of unchanged parts, and they would have a good similarity correlation with each other. So we could make D plus one and repeat the process, which means to divide the function into smaller parts. And we could find the matching function in X_2 based on the maximum value of similarity wave array. As Fig.6 shows, we could detect the matching function based on each coefficient method, but each method has its drawbacks. The cross-version function's vectors may be similar to each other, but its signal wave may not be linear correlated and vice versa. Thus, we take all these multiple results into account to detect the matching function in our method.

IV. EVALUATION

A. IMPLEMENTATION AND SETUP

We obtain the dataset from Alpha-Diff [21] and evaluate the proposed method concerning its search accuracy. First, we can make sure that the binaries' functions we used in the dataset[1] are identified correctly, which means all the functions f_{1i} and f_{2j} in binaries X_1 and X_2 can be determined. We implement the proposed method in MATLAB [22]. Our experiments are conducted on a PC with 3.6 GHz Intel CPU i7-4790 and 10 GB RAM.

Dataset: We get the large dataset used in the paper [21]. The dataset is a set of 2,489,793 positive samples (i.e., pairs of matching functions) from 66,823 pairs of cross-version binaries in the x86 Linux platform. There are two sources for the dataset. The first source is the Github repository. The dataset includes the source code from 31 projects with 9,419 releases. Each release is compiled with the compiler GCC with the default optimization options and each project's two successive releases of binaries were put into one pair. The second source is the Debian package repository, 895 packages with 1,842 versions were collected from the Ubuntu 12.04, 14.04 and 16.04 platform. Each version of binary with its closest version was grouped as a pair. For each pair of cross-version binaries, pairs of matching functions were retrieved to make sure they have the same name but are not identical. And to increase the diversity, some pairs of functions which are identical in cross-version binaries were extracted and added to the dataset. About 1.52 percents of pairs of cross-version functions in the dataset are identical. The test dataset in Alpha-Diff [21] is a disjoint subset of the whole dataset, including 9,308 pairs of binaries. And the author of the Alpha-Diff [21] split the testing set into a big subset and small subset. Each binary pair in the big subset contains more than 300 function pairs. However, they do not provide the test dataset.To truly and objectively test the effectiveness of the method, in the construction of the experimental test data set, we did not perform any manual sample selection operation but used the random selection method in the batch command in the Windows system. And the test data set contains 423,282 functions of cross-version matching functions, from 12,000 pairs of cross-version binaries. We also split the testing dataset into a big subset and a small subset. The small subset consists of 11,797 pairs of cross-version binaries, including 289,165 pairs of cross-version matching functions. The big subset consists of 203 pairs of cross-version binaries, including 134,117 pairs of matching functions.

[1] https://twelveand0.github.io/AlphaDiff-ASE2018-Appendix

B. EVALUATION METRIC

The goal of the method is to identify the matching function accurately. In the binary code similarity detection problem, TP(True positive) should be samples that are judged to be a correct matching function, FP(False positive) refers to samples that are not matching functions but are judged to be matching functions, and FN(False negative) are samples that should be detected as a matching function, but it is not correctly detected. In the experiments, every query has only one correct answer (matched function). Therefore, the number of FP and FN is the same, resulting in the value precision is the same as of the recall. And Inspired by the Alpha-Diff, we also evaluate the method by the evaluation metric $Recall@K$. The $Recall@K$ is defined as follows.

$$Recall@K\ (X_1, X_2) = \frac{\sum_{i=1}^{T} hit@K\ (f_{1i})}{T} \tag{7}$$

where X_1 is a binary including the functions f_{11}, f_{12}, $f_{13}, \ldots, f_{1n}$. X_2 is a binary including the functions f_{21}, f_{22}, $f_{23}, \ldots, f_{2m}$. We assume they have T pairs of matching functions, i.e.(f_{11}, f_{21}), (f_{12}, f_{22}), . . ., and (f_{1T}, f_{2T}). We denote the top K similar functions calculated by each algorithm and each part of functions as $topK(f_{1i})$ function set. And we denote $hit@K(f_{1i})$ as whether the matching functions of f_{1i} is in $topK(f_{1i})$. $hit@K(f_{1i})$ is defined as follows.

$$hit@K(f_{1i}) = \begin{cases} 1, & f_{2i} \in topK(f_{1i})\ and\ i \leq T \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

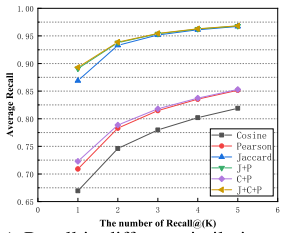
C. PARAMETERS IN THE METHOD

Besides, the method involves some parameters and design decisions, e.g. the similarity coefficient method, the length limit, different function partition number and the minimum byte size. The choices of these parameters could affect the performance of the method.

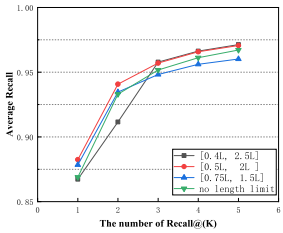
We have conducted a series of experiments to select proper parameters. Due to the time and resources limitation, we evaluate each model's performance on a subset of the testing set, including 29,168 function pairs.

1) SIMILARITY COEFFICIENT METHOD

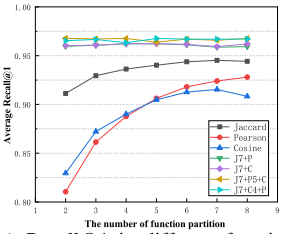
We have evaluated the performance of the method with different similarity coefficient methods, as shown in Fig.7(a). (In Fig.7 abbreviated the Jaccard coefficient method, Cosine similarity and Pearson correlation coefficient as J, C and P) We can see that the method performance is affected by the similarity coefficient method. The method based on different



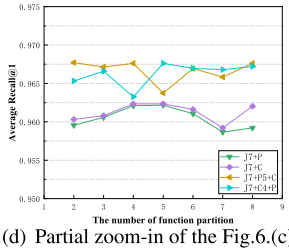
(a) Recall in different similarity coefficient method



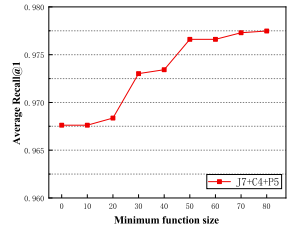
(b) Average recall in different length limit



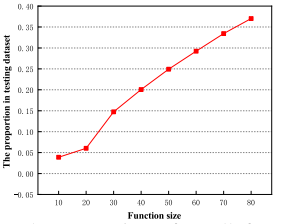
(c) Recall@1 in different function partition number



(d) Partial zoom-in of the Fig.6.(c)



(e) Recall@1 in different minimum function size



(f) The proportion of small functions in dataset

FIGURE 7. Evaluation of design parameters in the method.

similarity coefficient algorithms is complementary to some extent. And the method with multiple similarity coefficient algorithms performs better than any single one.

2) LENGTH LIMIT

We have evaluated the performance of the method with different function length limit, i.e., the proper length ratio of candidate functions in binary X_2 based on the length of the function f_{1i}, as shown in Fig.7(b). It is shown that if the length limit is set to $[0.5l, 2l]$, the method, in general, performs best. And in the experiment, the number of missing function pairs is only eighteen, which means the method still almost covers all the test samples. Thus, in the later test, we set the length limit to $[0.5l, 2l]$.

3) DIFFERENT FUNCTION PARTITION NUMBER

We have evaluated the performance of the method with different function partition number, as shown in Fig.7(c). The J7 + P means when the function partition number of Jaccard similarity coefficient is seven, the performance of the method with different function partition number of the Pearson correlation coefficient. The partial zoom-in of the Fig.7(c) is shown in Fig.7(d). It is shown that when the function partition number of the Jaccard similarity coefficient is seven, the function partition number of Cosine similarity is four and the function partition number of Pearson correlation coefficient is five, the method performs best. And the recall of our method is about 96.7%.

4) MINIMUM FUNCTION SIZE

Many similarity coefficient methods usually cannot get a good performance on small arrays–even minor changes of the array may cause a large influence on the similarity coefficient values. And too small bytes size might always cause the input part to have a high value of similarity coefficient with any other parts in X_2, which makes the matching functions difficult to be identified. So in the last, we have evaluated the performance of the method on different datasets, in which the length of the functions is large than different binary sizes, as shown in Fig.7(e). It is shown that the method performs better in the subset of the big functions. But the proportion of the small functions in the test dataset is shown as Fig.7(f). As we can see that many functions are short in length and we should not set the minimum function size too high. According to Fig.7(e) and Fig.7(f), the best value of the minimum functions size is 30 bytes. The method could cover almost 85 percent of the dataset and the recall could reach 97.3%.

D. ACCURACY ON TESTING SET

In the last, we evaluate the proposed method on the whole test dataset consisting of 12,000 pairs of cross-version binaries, including 423,282 positive samples and calculate the metric *Recall*@1 and *Recall*@5 for each function. Table 1 shows the average recall results.

TABLE 1. The recall accuracy of the proposed method on test dataset.

	Small dataset		Big subset	
	Whole dataset	Without small functions	Whole dataset	Without small functions
Recall@1	0.9663	0.9721	0.8822	0.8921
Recall@5	0.9834	0.9842	0.9354	0.9412

The Alpha-Diff solution [21] and our method are based on raw bytes of binaries for cross-version similar code detection. In almost the same test environment, the *Recall*@1 of Alpha-Diff is 0.953 on their test dataset and 0.885 on the big subset. And our method could reach 0.9663 on the test dataset and 0.8828 on the big subset. We successfully detect 397,746 samples on the test dataset in total, among which 279,422 samples on the small subset and 118,324 samples on big subset. The big subset consists of 134,117 pairs of matching functions from 203 pairs of binaries. The binary size in the big subset is between a few hundred Kilobytes and several Megabytes. And the performance of the solution on the large size binaries could be shown by testing the method on the big subset to some extent.

Although we do not compare our method with the alpha-diff solution in completely consistent conditions, the experimental results show that our method could also have a good effect in the field of cross-version binary code similarity detection. When there is a special situation that the problem cannot be solved by deep learning methods, it proves that the results detected by the method in this paper are also credible.

However, our method still cannot detect some function pairs in the dataset. After analyzing the wrong function pairs, we find three features of functions in the dataset that reduce accuracy.

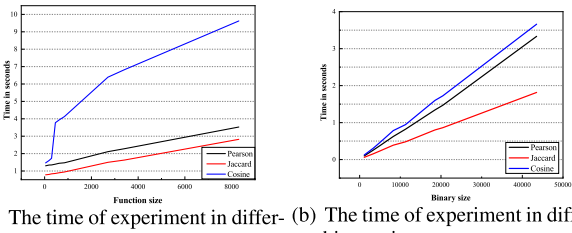
(1) In some binaries, the length and raw bytes of some different functions are similar which would cause a decrease in the accuracy.

(2) Some functions in the dataset have changed dramatically, causing a huge difference between the matching functions. And our method cannot detect these matching functions.

(3) Some instructions may have similar raw bytes while their semantic meanings are not similar. Besides, the raw bytes contain some ''noises'' like specific register values. These problems would decrease the accuracy of our method in the experiment.

E. EFFICIENCY

We also evaluate the efficiency of our method in solving the BCSD problem. For each binary, the length of the functions could range from dozens to thousands of bytes. So we test functions of different lengths to get the experiment time of the different similarity coefficient algorithm. Fig.8(a) shows the experiment time of different function size when the binary size is equal to 18,519 bytes. Fig.8(b) shows the experiment time of different binary sizes. In this experiment, the length of the functions we select for each binary is all range in [100, 200] bytes.



(a) The time of experiment in different function size

(b) The time of experiment in different binary size

FIGURE 8. Efficiency evaluation on test dataset.

From Fig.8, we can observe that the experiment time in general increases along with the binary size and function size. The method based on the Jaccard similarity coefficient is the most efficient of all. In the dataset, the length of most binaries is shorter than 20,000 bytes and the length of most functions is shorter than 1,000 bytes, which means in the experiment, our method could detect the matching function within 4 seconds of most samples. In the experiment on the big subset, after we set the function length limit, many functions that do not meet the requirements are excluded. The length of the sliding window experiment generally does not exceed 100,000 bytes and the time required for the method to run is about a few seconds. But in practice, if the sliding window experiment needs to be tested on a large file (more than 1MB), the running time of the sliding window algorithm would take about tens of seconds, which is inefficient. However, we could split the binary into multiple smaller sub-files and use a distributed algorithm to improve efficiency in future work. This can greatly improve the efficiency of the method.

V. DISCUSSION

Many advanced similarity detection solutions such as Bindiff, Gemini, Alpha-diff, etc are still the most important and best solutions currently. And to some extent, these solutions are better than the methods proposed in this paper in terms of overall performance(They could solve more difficult problems such as cross-platform and cross-compiler problem). However, one of the values of our method is that, from another research perspective, without relying on CFG, deep learning algorithm and other related attributes, we propose a new method that can solve the cross-version binary code similarity detection problem. Our method is not proposed to beat and replace various methods such as Bindiff, Gemini, Alpha-diff. Instead, it is used as one of the different methods to cope with various complex situations that exist in the real-world environment.

The technology of cross-version BCSD can be used to detect malicious code, (e.g. for some specific viruses). And this technology can also be used for vulnerability mining and plagiarism detection. Hackers can obtain adversarial samples by modifying few pixels in the image, making the deep learning model ineffective, but most of these methods do not affect the structural characteristics of the binaries, so they have less impact on the method proposed in this paper. There are many good pure syntactic solutions based on raw bytes, such as similarity detection based on opcodes. And they can be performed by Yara search or other tools. But in general, these methods and our proposed methods still have some different characteristics. 1) Relying on methods such as Yara and other search tools, most of them need to manually analyze some unique attributes of the application target, such as some signatures or other characteristics and then use these characteristics to perform the matching search. However, our method does not require manual analysis to extract features, but maps the code into vectors and signals, and searches in the binaries based on its linear correlation and other related attributes. The application is for all cross-version binaries in the same environment, and there is no need to extract different features for different binaries; 2) The features directly displayed in the opcode have some difference with the attribute characteristics that the code maps into vectors and signals. Whether using opcode or our method, we all could solve the cross-version similarity problem to some extent. But one of the key points and innovation of the method proposed in this paper is to solve the problem through the attribute characteristics of another code expression and provides a new idea for research in the fields of vulnerability mining, malware variant detection, plagiarism detection, etc. In practical applications, these methods are combined with each other, and the comprehensive application of various methods could often get better practical results.

VI. LIMITATIONS AND FUTURE WORK

Some instructions have the same semantic meaning while their raw bytes are significantly different. And this is very common among different compilers, different optimization options, and different platforms. The method proposed in this paper is still inadequate in many aspects, and in many ways is not better than most current solutions that rely on the CFG of binaries. How to deal with the difference of the raw bytes in cross-compilers and cross-platforms binaries is one of the shortages of our method and it is the focus of our future research. Similarly, some instructions may have similar raw bytes while their semantic meanings are not similar. In future work, we will conduct in-depth research based on this problem.

The signal processing based method is a novel method to detect the similarity of binary code, this approach does not work well when the length of the function is too short or the change of the functions is too big. And it also does not work well on small binary files with small functions. For small binaries, because the workload of analyzing binary is relatively small, we recommend using other methods, such as opcodes search or manual analysis to solve the problem. Besides, when we divide the functions into several parts, we can get a small range of functions which include the matching functions. But how to select the matching functions in the small range of functions to improve the effectiveness of the method still need more research. And future work possibly involves using a more advanced method to solve these problems.

The problem of the noises in raw bytes is an important issue that needs to be considered when doing binary code similarity detection. Due to the influence of this factor, the use of some simple features (such as directly extracting a certain part of the feature code, calculating the hash, average, partial average, etc.) for detection cannot get good results. Compared to more complex issues such as cross-architecture, cross-compiler and cross-compilation-option problems, there are relatively few noises in cross-version binary code. The method proposed in this paper is less affected by the noise in detecting similarity. However, for more complex code such as cross-architecture, cross-compiler and cross-compilation-option binary, the method proposed in this paper does not work very well for the time being. In further research, we will explore whether preprocessing methods such as intermediate languages can be used to solve the problems in the binary code.

VII. RELATED WORK

Most Previous works on binary code similarity detection are based on the control flow graphs (CFG) of functions and graph-isomorphism theory. Eschweiler *et al.* [8] proposed an approach to find similar functions in binary code, and they use the method to identify bugs in binaries. They propose a set of numeric features and employed a pre-filter based on the features to fast identify the candidate functions. Feng *et al.* [9] proposed a method to covert the CFGs into high-level feature vectors instead of directly using the raw features of binary code. They could search the vulnerability in a large set of the firmware images. These solutions are based on the graph comparing, which lacks the polynomial-time method and does not consider the semantics of assembly-level features.

Gao *et al.* [12] extend the graph isomorphism with symbolic execution to find semantic differences. Ming *et al.* [13] extend the GI theory with deep taint and input generation techniques to find semantic differences in CFGs. Chandramohan *et al.* [15] use an inline technique to capture the complete semantics features of functions and generate the candidate functions by using OS neutral function filtering. They extract the variants traces from the functions and detect the similarity of functions by machine learning methods. David *et al.* [16] divide the functions into smaller comparable fragments and detect the similarity of functions by the similarity between fragments.

Inspired by Feng *et al.* [9], Xu *et al.* [10] represent the functions as a control-flow graph with attributes (ACFG). They firstly use a neural network to generating embedding for functions. They used the Siamese network [23], which is adapted from Structure2vec network [24], to convert the ACFG into an embedding. However, the solution heavily relies on CFG features and block-level attributes. Gao *et al.* [17] present VulSeeker, a semantic learning-based vulnerability seeker for cross-platform binary. By integrating the CFG and the DFG of the binary function, they capture more function semantics and acquires a higher accuracy and efficiency.

Without relying on the CFG of function, Asm2Vec [18], INNEREYE [19] and SAFE [20] explore many new methods to compute the embedding vector of binary functions. Ding *et al.* [18] employ representation learning to construct a feature vector for assembly code. They only need assembly code as input and do not require any prior knowledge such as the correct mapping between assembly functions and provide more robustness to code obfuscation and compiler optimizations. Zuo *et al.* [19] regard instructions as words and basic blocks as sentences, and propose a novel cross-(assembly)-lingual deep learning approach to solve cross-architecture BCSD problem. They utilize word embedding and LSTM to automatically capture the semantics and dependencies of instructions and solve the BCSD problem among basic blocks. Liu *et al.* [21] also propose a method to use the neural network to extract the features from the functions. They extract the features directly from the raw bytes of functions, without any human bias. They use intra-function features, inter-function features and inter-module features to train a CNN network through the Siamese network, providing better accuracy. Massarelli *et al.* [20] propose SAFE, a novel architecture for the embedding of functions based on a self-attentive neural network. It works directly on disassembled binary functions, does not require manual feature extraction. And it is computationally more efficient and is more general as it works on stripped binaries and multiple architectures.

There is a good deal of related research on the detection of similarity between signals in the field of signal processing. Hu *et al.* [25] propose a new method to detect the correlated alarms. They quantify the correlation level based on the Pearson correlation coefficient and show their method perform better in detection correlated alarms and uncorrelated ones than existing methods. Bertin *et al.* [26] also use the Pearson correlation coefficient to predict the wave energy resources development. They calculate the PCC between the North Atlantic Oscillation and North Atlantic significant wave height and show that the North Atlantic Oscillation partially controls the interannual variability of significant wave height.

Cheng and Zhang [27] propose a Jaccard Coefficient-based Bi-clustering and Fusion method for recommender systems. They use the density peak clustering method to cluster the user-item rating matrix and estimate the missing values for sparsity data to cope with the sparsity problem in cold-start settings. The experiment shows that their approach can improve the performance of user recommendations at the extreme levels of sparsity in the user-item rating matrix. Dharavath and Singh [28] propose an efficient integrated solution to the entity resolution problem based on the Jaccard similarity coefficient. From the experiments on three citation databases evaluate the contributions of the Jaccard similarity coefficient and Markov logic, we can see that a few rules in Markov logic and Jaccard similarity give more efficient results. Plansangket and Gan [29] proposes a query suggestion method combining two ranked retrieval methods: TF-IDF and Jaccard coefficient. They evaluate the method using several performance criteria and users' judgment as well in terms of the quality of the generated query suggestions and the result shows that their method could improve the relevance of the returned documents in interactive web search. The above researches show that the Pearson correlation coefficient and Jaccard coefficient are well metric to quantify the similarity and correlation between signals.

VIII. CONCLUSION

In this paper, we propose a novel and lightweight method for solving the cross-version BCSD problem. The method does not need much recourse and could be easily deployed on PCs and other lightweight devices. Without relying on CFG and deep learning algorithms, the method extracts the function features directly from the raw bytes of binaries. And together with the signal processing technique, the method performs well on the cross-version BCSD problem. To the best of our knowledge, there exist no previous works which apply signal processing techniques to cross-version BCSD problems. The method would be proof that the signal processing technique is suitable for solving the BCSD problem. However, there are still a lot of problems in the research of detecting similarity in cross-version binaries. We hope that this work could serve as an inspiration for more research.

ACKNOWLEDGMENTS

The authors are very thankful for Bintao Yuan, Xianda Zhao, Peiwu Dai and Wenli Zhu for their help in the preparation of the experiment and paper review. They thank the anonymous reviewers for their detailed comments which helped to improve the quality of the paper.

REFERENCES

[1] 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.

[2] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proc. 22nd USENIX Secur. Symp. (USENIX Secur.)*, 2013, pp. 81–96.

[3] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proc. NDSS*, vol. 9, 2009, pp. 8–11.

[4] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1157–1177, Dec. 2017.

[5] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 117–128.

[6] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2008, pp. 143–157.

[7] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, "SPAIN: Security patch analysis for binaries towards understanding the pain and pills," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. (ICSE)*, May 2017, pp. 462–472.

[8] 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.

[9] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2016, pp. 480–491.

[10] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.

[11] Zynamics. *Bindiff*. Accessed: Dec. 16, 2018. [Online]. Available: https://www.zynamics.com/bindiff.html

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] 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

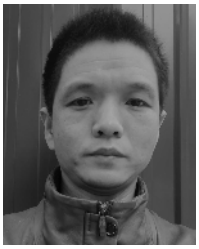
[21] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αDiff: Cross-version binary code similarity detection with DNN," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng. (ASE)*, 2018, pp. 667–678.
[22] Mathworks. *Matlab 2014*. Accessed: Dec. 17, 2018. [Online]. Available: https://www.mathworks.com/products/matlab.html
[23] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, "Signature verification using a 'siamese' time delay neural network," in *Proc. Adv. Neural Inf. Process. Syst.*, 1994, pp. 737–744.
[24] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2702–2711.
[25] W. Hu, J. Wang, and T. Chen, "A new method to detect and quantify correlated alarms with occurrence delays," *Comput. Chem. Eng.*, vol. 80, pp. 189–198, Sep. 2015.
[26] X. Bertin, E. Prouteau, and C. Letetrel, "A significant increase in wave height in the North Atlantic Ocean over the 20th century," *Global Planet. Change*, vol. 106, pp. 77–83, Jul. 2013.
[27] J. Cheng and L. Zhang, "Jaccard coefficient-based bi-clustering and fusion recommender system for solving data sparsity," in *Proc. Pacific–Asia Conf. Knowl. Discovery Data Mining*. Cham, Switzerland: Springer, 2019, pp. 369–380.
[28] R. Dharavath and A. K. Singh, "Entity resolution-based Jaccard similarity coefficient for heterogeneous distributed databases," in *Proc. 2nd Int. Conf. Comput. Commun. Technol.* New Delhi, India: Springer, 2016, pp. 497–507.
[29] S. Plansangket and J. Q. Gan, "A query suggestion method combining TF-IDF and Jaccard coefficient for interactive Web search," *Artif. Intell. Res.*, vol. 4, no. 2, pp. 1–7, Aug. 2015.

HUI GUO received the master's degree from the Electronic Engineering Institute in 2017. He is currently pursuing the Ph.D. degree with the College of Electronic Engineering, National University of Defense Technology, Hefei, China. His research interests include network security, malware detection, vulnerability discovery, and deep learning.

SHUGUANG HUANG received the Ph.D. degree from the University of Science and Technology of China, Hefei, China. He is currently a Professor with the College of Electronic Engineering, National University of Defense Technology. His research interests include computer science, computer engineering, and network security.

CHENG HUANG (Member, IEEE) received the Ph.D. degree from Sichuan University. He is currently an Assistant Research Professor with Sichuan University. His research interests include cyber security, attack detection, and threat intelligence analysis.

MIN ZHANG received the Ph.D. degree from Anhui University. He is currently a Professor with the National University of Defense Technology. His research interests include machine learning, data mining, and computer security.



ZULIE PAN received the Ph.D. degree from the Electronic Engineering Institute, Hefei, China. He is currently a Professor with the College of Electronic Engineering, National University of Defense Technology. His research interests include vulnerability discovery, network security, and computer science.

FAN SHI received the master's degree from the Electronic Engineering Institute, Hefei, China. He is currently an Associate Professor with the College of Electronic Engineering, National University of Defense Technology. His research interests include data analysis, network security, and networking protocol analysis.

HUI HUANG received the Ph.D. degree from the Electronic Engineering Institute, Hefei, China. He is currently a Teacher with the College of Electronic Engineering, National University of Defense Technology. His research interests include vulnerability discovery, network security, and computer science.

DONGHUI HU (Member, IEEE) received the Ph.D. degree from Wuhan University, Wuhan, China, in 2010. He is currently an Associate Professor with the School of Computer Science and Information Engineering, Hefei University of Technology. His research interests include steganography analysis, network security, and digital image forensics.

XIAOPING WANG received the master's degree from Anhui University. She is currently an Associate Professor with the College of Electronic Engineering, National University of Defense Technology. Her research interests include computer science and network security.