RouAlign: Cross-Version Function Alignment and Routine Recovery with Graphlet Edge Embedding

Can Yang, Jian Liu, Mengxia Luo, Xiaorui Gong, Baoxu Liu

RouAlign: Cross-Version Function Alignment and Routine Recovery with Graphlet Edge Embedding

Can Yang[1,2], Jian Liu[1,2], Mengxia Luo[1,2], Xiaorui Gong[1,2], and Baoxu Liu[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, China
{yangcan,liujian6}@iie.ac.cn

Abstract. Reverse engineering is labor-intensive work to understand the inner implementation of a program, and is necessary for malware analysis, vulnerability hunting, etc. Cross-version function identification and subroutine matching would greatly release manpower by indicating the known parts coming from different binary programs. Existing approaches mainly focus on function recognition ignoring the recovery of the relationships between functions, which makes the researchers hard to locate the calling routine they are interested in.

In this paper, we propose a method using graphlet edge embedding to abstract high-level topology features of function call graphs and recover the relationships between functions. With the recovery of function relationships, we reconstruct the calling routine of the program and then infer the specific functions in it. We implement a prototype model called RouAlign, which can automatically align the trunk routine of assembly codes. We evaluated RouAlign on 65 groups of real-world programs, with over two million functions. RouAlign outperforms state-of-the-art binary comparing solutions by over 35% with a high precision of 92% on average in pairwise function recognition.

Keywords: Edge Embedding · Calling Routine Recovery.

1 Introduction

An essential purpose of reverse engineering is to pick out known calling routines from a new binary program. This analyzing task is generally used in malware family classification, reused component detection, patch comparison, and so on. But it could be a tedious job to find out every mutation of a program, especially when the main functionality stays the same but some calling routines added or removed. What's more, it would encounter plenty of difficulties in cross-architecture, cross-OS, cross-compiler, and cross-optimization programs routine cognition. This problem has now become more crucial while Internet-of-Things (IoTs) become massive and fragmentated.

For instance, Figure.1.A shows a message processing routine of a printer with CVE-2017-2741. Now, if we have obtained the function call graph (Figure.1.B) of another printer (with different architecture here), analyzing is needed to verify the vulnerability. In this case, traditional tools [9] could hardly help. A common way in practice is to locate the context functions by cross-references firstly. Then trace the calling routine to see whether a similar function exists in Figure.1.B. We note that only a subset of functions in the routine are concerned by researchers, and the connectivity between functions is important in the analysis.

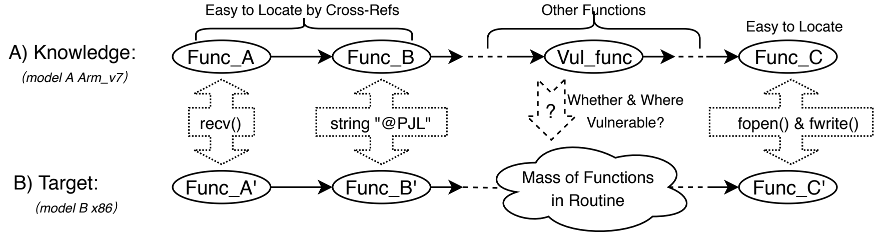

Fig. 1: Motivation: Reuse Knowledge of (A) to Search Vulnerabilities in (B)

Actually, the same vulnerability might be shared by dozens of binaries from different products. But automatically analyzing these cross-version binaries is complicated for many reasons. Generally speaking, difficulties are: 1) Different compilers prefer difference memory arrangements and inline hobbits. 2) Different compiler optimizations and obfuscators would greatly change the control flow [11,19]. 3) Library and system functions vary a lot within different operating systems. 4) A software might change a lot after years of development, even if its main features were hardly modified.

Existing state-of-the-art solutions for cross-version binary analysis are not routine-sensitive. Approaches [7,14,15,21,26] are mainly trying hard to understand the semantics of functions for identification. These methods tend to identify functions directly, but isolated. Approaches [8,17,20,24] take the Function Call Graphs (FCG) into consideration. But these methods only utilize intuitive features like degrees of nodes, as assistance to the function internal features. However, considering in case cross-version analysis, internal features are not stable and reliable. We design a method to better utilize FCG than ever. By abstracting higher-order structural features into vectors and then modeling them via neural networks, we show that the FCG could play an important role in routine recovery and function alignment.

Our Approach: Alignment. In this paper, we aim at finding a way to recognize a common calling routine between cross-version binaries, and then identify functions in it. For the purpose of recognizing the calling routine, we must have abilities to recover the relationships between functions (mainly caller-callee relationship). With the recognized relationships, reverse engineers can pick out a calling routine from the FCG of an unknown program. Once the calling routine is aligned with a known routine, the functions in the same position can be viewed as functional equal. This method is what we called *Alignment* method because it infers a function by its position in a routine rather than investigate the inner implementation of the function. This method should be robust enough against variations such as function inline and addition function calls.

For relationships recovery, we propose a new algorithm based on Graphlet Edge Embedding (described in Section 3.3). This is inspired by the ideas of Graphlet Degree Signatures [22], which is a famous alignment algorithm in bioinformatics. Our method starts from a simple assumption that the similar functions in program have similar graph structures in the function call routine, and reasonable modifications (function inline, additional calls, etc., which could be introduced by programmers or compilers) to the graph structure is recognizable. For instance, a distributor function usually follows the input routines and has many subroutines for special tasks. In short, we used edge embeddings to abstract high-level features of functions in function call graphs, and then recover the caller-callee relationships between functions to reconstruct the function calling routine. With this method, we can tell a known routine from a new binary, thus functions in the routine can be recognized and an overall knowledge of the program can be achieved.

We have implemented a prototype tool called RouAlign. RouAlign utilized graphlet edge embeddings to align two calling routines automatically and then identified functions by the aligned positions in the routines. We evaluated RouAlign in 65 groups of cross-version binaries with over 200,000 functions. And we compared the function recognition results of RouAlign with the results of BinDiff and the results of Gemini. RouAlign performed better than both and showed a great potential of function call graphs in function recognition.

In summary, our contributions are as follows.

- **Routine Recovery.** We present a multistage approach to align FCGs where the calling routines are preserved as much as possible.
- **Edge Embedding Algorithm.** We propose a method to embedding edges in FCGs, and an algorithm to recover caller-callee relationships between functions with abilities to distinguish modifications like function inline.
- **Scalable Design.** Compared to existing algorithms, our algorithm is parallelable (the procedure can be speeded up via multi-processing) and incrementable (the results would be expanded when given extra knowledge) besides high precision.
- **Better Performance.** Compared to popular commercial binary diffing tools, our prototype can perform 35% better precision and 25% better recall on average within cross-version binaries.

2 Problems and Challenges

In this section, we address the problem to study, challenges facing, and reasonable ways to solve them. Some important symbols are defined as well.

2.1 Function Alignment

Existing function matching methods can be roughly classified into two categories. One is pairwise matching, which searches common functions in a pair of programs. The other is library matching, which represents functions in brief forms (i.e., embedding) and searches a new function from a pre-built representation library. *Function alignment* is a kind of pairwise matching, but differs from function matching methods. Function matching methods aim to find two functions exactly the same. In contrast, function alignment tries to find functions of the same use (namely, functionally equal).

Definition 1. *Functional Equal: If function F_a and F_b take the same responsibility in their individual calling routines, they are functional equal. Ideally, functional equal means F_a and F_b can be substituted with each other in their calling routines.*

Considering an example of algorithmic upgrade, suppose an old program A has a routine: "$Input \rightarrow Encrypt \rightarrow MD5 \rightarrow Output$". And a new program B has a routine: "$Input \rightarrow Encrypt \rightarrow SHA1 \rightarrow Output$". In both program A and B, the *Input*, *Output* and *Encrypt* are exactly the same. Usually, Function matching methods [7,15] will mark the $SHA1$ and $MD5$ as different and the relationships with both *Input* and *Output* are ignored. But function alignment should mark that $SHA1$ and $MD5$ are of the same use (both are hash functions) and the calling routine stays successive. This makes alignment methods robust when figuring out the main routine of a program, especially when some components were changed. We regard alignment as a relationships-defined procedure, and formally define the function alignment as follows.

Definition 2. *Function Alignment: given a target program P_{trgt} and a template program P_{tmpl}, function alignment aims to find a function mapping $A(F_{tmpl}) \rightarrow F_{trgt}$ between FCGs from P_{tmpl} to P_{trgt}, where F_{trgt} are functional equal to F_{tmpl}.*

In case the functions in P_{tmpl} were known, the function alignment procedure could be viewed as knowledge reasoning from P_{tmpl} to P_{trgt}. In Biological Network Alignments, it has been shown that aligned networks are functional similar [18]. Our experiments show that rule also worked for Function Call Graph Alignments.

2.2 Challenges

Directed Heterogeneous Network. Network alignment and graph alignment are also hot but tough topics in literature. In reverse engineering, FCG provides limited information. Data refers, syscalls and many attributes of function are not negligible. Taking this into consideration, the FCG becomes a heterogeneous network. This means, for reliable and convincing performance, our function alignment problem might be equivalent to heterogeneous network alignment problems. Although heterogeneous networks preserve richer information than homogeneous networks, they face more challenges [4]. In addition, the development of algorithms on heterogeneous networks is not mature enough now.

Low Recognizability of Sparse network. The network alignment algorithms used in other fields, such as bioinformatics and graph theory, are usually designed for denser networks. However, in function call graphs, most functions only relate to other few functions. This means if we naively represent a function as its surrounding topologic, many functions might not be distinguishable. This is one of main reasons why embedding methods, like structure2vec [1,16], leave branch of functions as "similar". Low precision at one will significantly make the alignment methods not reliable because the mistakes would propagate via relationships.

Brief Solution. To overcome these challenges, we adapt two novel methods. Firstly, we separated the heterogeneous network into two layers. One contains the relationship between functions and other attributes; the other is a directed homogeneous network where nodes are functions. Secondly, instead of recognizing the nodes directly, we take a detour to recover edges among nodes, and design a new method to evaluate the similarity of these edges.

3 Function Alignment Method

In this section, we first show the overall workflow of RouAlign, which recover two informative structures from binary codes as necessary data. Then we introduce the core process of graphlet edge embedding method to extract higher-order graphic features. And finally, we describe the approach to aligning functions and recovering calling routines.

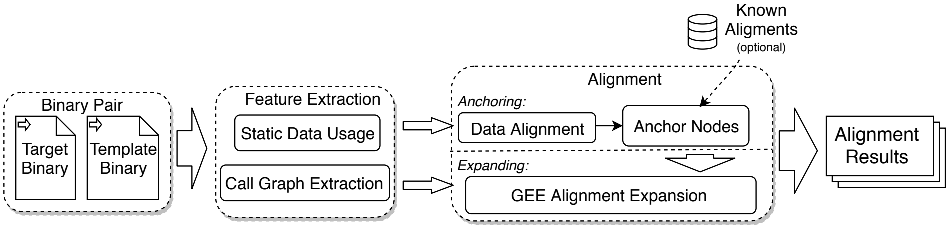

Fig. 2: Overview of RouAlign

3.1 Overview

The RouAlign is designed to align calling routines from the target binary to the template binary. Figure.2 shows the whole process of the tool. The first step is extracting necessary information from binaries, including static data references and function call graph. Static data includes string constants, numeric constants, etc. In experiments [12,13,14], they are proved to be reliable across versions. The function call graph preserves function nodes as well as caller-callee relationships between them. Thus, after extraction, our heterogeneous relation network has two layers with two different node types, one is func-data layer, the other is func-func layer.

The alignment stage can be separated into two phases — *anchoring* and *expanding* — for different layers of the relation network. The *anchoring* procedure tries to find some function nodes that are highly similar in the func-data layer. The *expanding* procedure trys to align more nodes from the anchored nodes in the func-func layer. They would be discussed later in section 3.2 and 3.3 . The basic idea of this method is to find some reliable nodes and then propagate the confidence as much as we can. This refers to the "seed and expand" idea of BLAST [3], and is also sort of simulation of the human analyzing procedure.

3.2 Anchor Nodes Searching

Many human researchers start to analyze a binary from limited entries, such as main functions, special library functions, functions with special and unique constants, etc. So inspired RouAlign. The reason behind is that, these features are the most likely to stay constant in the mutations [13]. These functions are naturally aligned and we call them anchor nodes.

We defined two kinds of anchor nodes. One comes from the running mechanism of executable binaries. In most situations, library calls and syscalls are explicit. For instance, Windows PE files use IAT to locate the library functions, and Linux ELF files use PLT. Many modern disassemble tools like IDA PRO can automatically indicate these calls in the binary. For this kind of anchor nodes, we directly take and use them. The other kind comes from human experiences. There are some unique constants in programs, such as s-Boxes in cryptography, magic bytes of protocols. Uses of unique constants can determine a function with high probability. Searching is needed for the second kind of anchor nodes.

Firstly, we match data nodes from different binaries, in order to find the constant data used by both two binaries. We extract some additional attributes for the data nodes, listing in the Table.1. We adapted SimHash [27] to resist slight changes to the constants. And then, we give these features to a linear classifier to tell whether two data are the same.

Table 1: Attributes for Data Nodes

Attribute Name	Weights
Length of Data	0.25
MD5 of Data	0.48
SimHash of Data	0.25
Offset in The Segment	0.02

*Weights referred to [14]

Secondly, after the data nodes from different binaries are matched, we use the TF-IDF (Term Frequency-Inverse Document Frequency) model to tell whether the function nodes related to the matched data nodes should be anchored. The TF-IDF model, which is a well-studied algorithm, can reflect how important the data is to a function. We pick out the function nodes with high weights and calculate the cosine similarity between them. The function nodes with high similarity are the anchor nodes that we want.

3.3 Expanding with Graphlet Edge Embedding

Nodes anchoring can align very limited part of the program, in most case. The expanding stage starts from the anchor nodes and expands the alignment alone edges. We introduced a heuristic method to evaluate the similarity between different edges. With the ability to match edges, we could find out the relationships between functions, and the calling routine could be recovered.

Graphlet Edge Signature. Respectfully, we name our new design "Graphlet Edge Signature", a method to characterize edges in directed graphs. Our design refers to the Graphlet Degree Signature [22] (a.k.a. GSV), which has been proven to be a successful design to extract topology structure in bioinformatics [18]. However, GSV was designed for node identification in an undirected graph. The Function Call Graph is a directed graph, which means the GSV should be redesigned. In practice, a reverse engineer can easily infer the unknown functions near a known one by FCG, especially by caller-callee relationships. Thus, it's reasonable to pay attention to edges recovery than directly node identification.

Firstly, we use a node pair $\langle N_c, N_t \rangle$ to represent a directed edge between center node (N_c) and target node (N_t). Then, we pick out all 2nd order neighbors of N_c and N_t. A graphlet is then defined by these nodes. In our design, we only concern about the motifs that related to 1st order similarity and 2nd order similarity. Motifs are recurrent and statistically significant subgraphs [23]. Using motifs makes us can focus on specific commonality once in a time. In our design, we only concern about the motifs that related to 1st order similarity and 2nd order similarity. We pick out 45 basic motifs, denoted as $m_1, m_2, \ldots, m_{45}$ in Figure.3. After that, we can count the motifs the target edge touches and get a vector V. Each dimension in the V stands for the number of times the motif m_i appearing in the graphlet. This vector is what we call the signature of the edge in the chosen graphlet (Graphlet Edge Signature, GES).

The policy of selecting these motifs in Figure.3 is not to enumerate all isomorphism subgraphs in the extracted graphlet, but to focus on the special relationships between the center edge and its surrounding edges. In addition, the chosen motifs should be able to compose any other complex graphlets. We view the in-and-out edges pair between two nodes as a bi-directed edge. These bi-directed edges stand for some highly recognizable relationships between functions like iterations and loops. And when counting rings, we do not distinguish the source point and the destination point to reduce the complexity. We focus on surrounding nodes no more than 2nd order, although the edge signature vector will be defined more precisely with higher-order neighbors. This is because the complexity and time consumption will increase sharply while going into higher-order relationships [24].

Distance Measurement. Now we have managed to represent a directed edge into the numeric format. A measuring system is needed for further usage. For distance measurement, there is a simplified, intuitive mathematical solution adapting from GSV. We define the distance D between the target edge E_{trgt} and

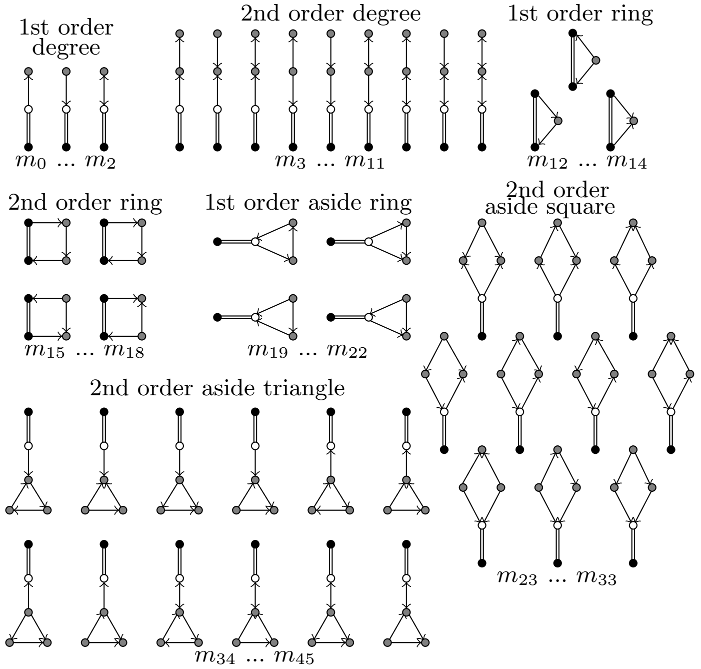

Fig. 3: Chosen Basic Motifs, in which central nodes, target nodes, and surrounding nodes are depicted as “black”, “white” and “gray” vertexes respectively.

template edge E_{tmpl} as below. The D_i is the distance at the i_{th} motifs. The E^i is the i_{th} member of the signature vector, meaning the number of times edge E touches motif m_i.

$$D = \sum_{i=0}^{45} D_i(E_{trgt}, E_{tmpl}) = \sum_{i=0}^{45} \frac{|log(E^i_{trgt}+1) - log(E^i_{tmpl}+1)|}{log(max\{E^i_{trgt}, E^i_{tmpl}\}+2)} \tag{1}$$

Obviously, the distance D results within range [0, 1). A smaller distance indicates there is more similarity between two edges. However, in most FCGs, the induced graphlet is usually not dense enough to contain most motifs above, leaving a lot of zeroes in the signature vector and leading to the distribution of distances close to 0. We can remove the part that has no value to revise the weights and remap the distribution of distances into [0, 1) — to ignore the similarity comes from common deficiencies. The distance can be now defined as:

$$D = \frac{\sum_{i=0}^{45} D_i}{\sum_{i=0}^{45} B_i} \tag{2}$$

$$B_i = \begin{cases} 1, E^i_{trgt} \neq 0 \ and \ E^i_{tmpl} \neq 0 \\ 0, others \end{cases} \tag{3}$$

Embedding. The method introduced above is experience-based and only suitable for distance measurement. Additionally, we introduce an embedding representation not only suitable for similarity measurement but also semantic preserved. We use a neural network encoder with the Siamese architecture [2] to generate the embedding of an edge.

The Siamese architecture uses two identical embedding neural networks. In our case, a 3-layer neural network was used. Each embedding network takes a GES vector V as input and outputs what we call Graphlet Edge Embeddings (GEE). GES vector V is a 45-dimensional vector as mentioned above. The final output of the Siamese architecture is the Euclidean distance of the two embeddings. While training, distances of similar input pairs were set to 0, and distances of dissimilar input pairs were set to 1. We formulate the Siamese network output distance D' for each input pair as:

$$D' = \|Embed(V_1) - Embed(V_2)\| \tag{4}$$

3.4 Inline Recognition

Our method is naturally suitable for inline recognition. Showing in Figure.4, the function `ingroup` was inlined into `check_suid` due to compiled differences. However, in the origin FCG (at left), if we connect a virtual edge from the caller of the inlined function directly to the callee of the inlined function (depicted as two dotted lines from `check_suid` to `bb_internal_get_grgid` and `bb_internal_getpwnam` in the figure), the signature vector V of the virtual edges would be quite similar to those edges in the inlined FCG. An addition of a function can be detected in a similar way. With this trick, expanding procedure can step over slight reasonable changes and perform more robust in calling routine alignment.

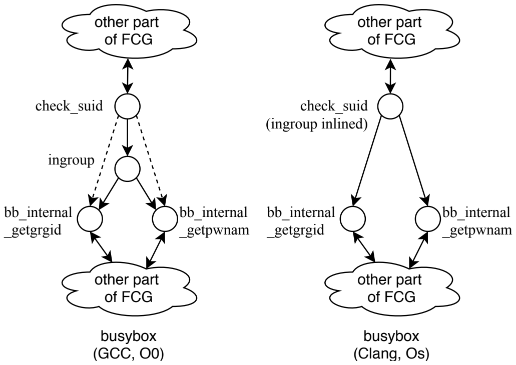

Fig. 4: Inline Recognition

The expanding procedure is now easy to explain: $1st$ choose an aligned node pair $\langle N_{trgt}, N_{tmpl}\rangle$. $2nd$ enumerate then embed edges of each node to obtain two sets of embeddings S_{tgrt} and S_{tmpl}. $3rd$ align edges pairwisely between the two sets by calculating the distances D (the average of both distances mentioned above). We leave edges "far from" any other edges alone, and then perform

Algorithm 1 Expanding Routine

```
Require: Aligned pair: $\langle N_{trgt}, N_{tmpl} \rangle$
  $S_{tgrt} \leftarrow GES(Edge)$, for all Edge connected to $N_{trgt}$
  $S_{tmpl} \leftarrow GES(Edge)$, for all Edge connected to $N_{tmpl}$
  repeat
    for all $V_1$ in $S_{tgrt}$, $V_2$ in $S_{tmpl}$ do
      if D($V_1$, $V_2$) < Threshold then
        mark the another two endpoints of the two edges as Aligned
      end if
    end for
    for all $V_1$ in $S_{tgrt}$, $V_2$ in $S_{tmpl}$ without alignment do
      Perform inline search
    end for
  until All aligned nodes have been expanded.
```

inline recognition on these isolated edges. The overall algorithm for expanding stage is summarized in Algorithm.1. Due to the space limit, we omit some details here, such as judgement of the closest distance and removal of duplications.

4 Evaluation

4.1 Implementation and Datasets

We had implemented a prototype of RouAlign. We used the IDA PRO as the tool to extract the necessary information (i.e., constant data, library functions, and the FCG) to construct the relation networks. We implemented the whole alignment stage with python, including isomorphic judgment and GEE algorithm. For embedding network training, we undersampled about 20 million edge pairs from the datasets. The embedding size was set to 20 empirically. Our experiments were conducted on a laptop with 8 GB memories and 4 cores at 2.6 GHz.

Our evaluation was base on two datasets: The first one was called the **horizontal dataset** where the binaries were compiled from the same source code but with different compilers and optimizations. This dataset contained 50 groups of binaries from 5 different programs. The second one was called the **longitudinal dataset** where the binaries were compiled from different source codes, and these source codes were referred to different versions during the development of the same software. This dataset contained 15 groups of binaries from 3 different generations of OpenSSL.

Each group needs at least 4 binaries: a tripped target, a stripped template, an unstripped target, and an unstripped template. The two unstripped binaries were used for ground truth extraction. With the debug symbols, the easiest way to get the ground truth of similar function pairs is comparing the function name. Some function name might change due to compiling definitions and developments. For example, The `OPENSSL_strlcpy` in OpenSSL 1.1.1 has a different name `BUF_strlcpy` in OpenSSL 0.9.8. We manually corrected these functions as a complement to the ground truth.

We use the Precision and the Recall metric. For every aligned function pair, if the pair is not in the ground truth, we count the precision as zero. For every function pair in the ground truth, if the pair is not in alignment results, we count the recall as zero. Therefore, the precision captures the ratio of function pairs that are correctly found, and the recall captures the ratio of function pairs that are supposed to be found.

4.2 Horizontal Comparison: same source, different compilation

Horizontal experiments were designed to simulate the circumstances where the same source code was reused in different environments. We implemented the horizontal experiment on 5 frequent-used real-world program projects. Each program was compiled into 5 different versions with different compile options. The compiler we used is `GCC` and `clang`. The options we used is O0 (without optimizations) Os (size-first optimization) and O3 (speed-first optimization). Binaries were aligned with each other by both RouAlign and BinDiff. BinDiff was used as the benchmark in our experiments. Table.2 shows the results of one-fifth of our horizontal experiments.

Table 2: **Horizontal Comparing Result of BusyBox.**

Binary Pairs	**Precision**		**Recall**	
	RouAlign	BinDiff	RouAlign	BinDiff
G$^{@}$O0 - G$^{@}$O3	0.928	0.651	0.770	0.229
G$^{@}$O0 - G$^{@}$Os	0.946	0.546	0.792	0.456
G$^{@}$O0 - C$^{@}$O0	0.990	0.765	0.874	0.623
G$^{@}$O0 - C$^{@}$Os	0.913	0.452	0.703	0.344
G$^{@}$O3 - G$^{@}$Os	0.991	0.945	0.926	0.336
G$^{@}$O3 - C$^{@}$O0	0.918	0.617	0.707	0.217
G$^{@}$O3 - C$^{@}$Os	0.953	0.653	0.755	0.245
G$^{@}$Os - C$^{@}$O0	0.937	0.506	0.723	0.419
G$^{@}$Os - C$^{@}$Os	0.961	0.593	0.775	0.459
C$^{@}$O0 - C$^{@}$Os	0.933	0.524	0.756	0.404

G stands for the **GCC** compiler and **C** stands for **Clang**.
$^{@}$***** indicates the compiler optimizations.
For instance, **G$^{@}$O0** means a binary compiled using **GCC** with option "**-O0**".

A case study on results in Table.2 shows that the precision of RouAlign is much higher than that of BinDiff, and the performance is very stable. Generally, the O0-Os and O0-O3 shows the lowest performance, because the compiler would introduce a lot changes to the original FCG on specific purpose. An interesting phenomenon is that, the more the version varies from each other, the better RouAlign performs than BinDiff. This is because traditional matching methods rely much on internal function features, which change a lot during compiling procedures.

We made statistics on all results of 5 different binary sets, and cited some results of other state-of-art solutions, showing in Table.3. In the first part of the table, we presented some average numbers of some important indicators to describe the datasets. The number of functions and the number of ground truth showed the scale of the binary. The number of anchor nodes showed how efficient the expanding procedure was. It's easy to summarize from Table.3 that the RouAlign could perform better on larger and more complex binaries, where the

Table 3: Horizontal Comparing Results Statistics.

	minigzip	**BusyBox**	**ImageMagick**	**OpenSSL**	**Sqlite3**
			Average Numbers		
Ground Truth	162.5	4711.6	4571.8	8105.2	2657.4
Total Functions	211.8	5200.0	5591.0	9054.6	3151.0
Extracted Datas	75.2	7946.2	16407.8	12822.4	3491.8
Symbolic Func.	26.3	29.0	455.0	135.1	117.6
Anchor Nodes	35.7	913.2	1254.7	546.3	346.0
			Precision		
RouAlign	0.939	0.946	0.900	0.941	0.788
BinDiff	0.733	0.625	0.433	0.336	0.434
Gemini[1]	0.750	0.828	0.397	0.546	0.454
asm2vec[2]	-	0.856	0.837	0.792	0.776
αDiff[3]	0.546	0.546	0.546	0.546	0.546
			Recall		
RouAlign	0.448	0.778	0.593	0.624	0.593
BinDiff	0.522	0.373	0.320	0.218	0.320
Gemini[1]	0.017	0.106	0.012	0.156	0.142

1: Gemini[1] here is a optimized version for the original program couldn't be accomplished in a 16GB memory machine. Gemini* was only tested on binary pair Clang-Os to GCC-O0)
2: Results of Asm2Vec[2] were cited (pairwise comparison on GCC-O0 and GCC-O3).
3: Results of αDiff is cited and then averaged of all their x86_64 results.

FCGs are more tremendous and the graphlet features are more representative. All these horizontal comparing results proved that relationships are useful and our method could handle the problem correctly.

4.3 Longitudinal Comparison: same software, different versions

The longitudinal comparison was designed to simulate the circumstances where the same program itself varies a lot from version to version. We chose 3 different source code versions of OpenSSL, among which time spans nearly 10 years and 3 big generations, and then compiled them with 5 different options.

As shown in Table.4, RouAlign performed much more stable among different compiler optimizations with high precision. The recall was improved a little over that of BinDiff, because the calling routine did change quite a lot during long-term iterations. It should be noted that the results of RouAlign are continuous with high precision, providing more powerful assistant to human researchers. All these longitudinal comparing results proved that our methods are usable in detecting long-term changes to binary and can still recognize calling routines in a high precision.

5 Limitations

Major limitation of alignment methods is the missing of many internal function features. We designed so in order to show that the FCGs could provide much information for understanding binaries. A better way we advised in the future is to combine RouAlign with some function embedding methods that could nicely represent internal function feature cross-versions.

Table 4: Longitudinal Comparing for Cross-Version OpenSSL.

Versions	**Precision**					
	Ver.100 to Ver.098		Ver.111 to Ver.100		Ver.111 to Ver.098	
	RouAlign	BinDiff	RouAlign	BinDiff	RouAlign	BinDiff
GCC-O0	0.953	0.843	0.789	0.493	0.692	0.436
GCC-O3	0.944	0.794	0.602	0.332	0.501	0.296
GCC-Os	0.950	0.824	0.781	0.439	0.674	0.377
Clang-O0	0.959	0.822	0.775	0.496	0.697	0.423
Clang-Os	0.956	0.796	0.779	0.432	0.669	0.366
	Recall					
	Ver.100 to Ver.098		Ver.111 to Ver.100		Ver.111 to Ver.098	
	RouAlign	BinDiff	RouAlign	BinDiff	RouAlign	BinDiff
GCC-O0	0.675	0.676	0.401	0.406	0.311	0.352
GCC-O3	0.618	0.544	0.231	0.228	0.185	0.200
GCC-Os	0.650	0.605	0.422	0.326	0.309	0.278
Clang-O0	0.671	0.584	0.385	0.357	0.304	0.305
Clang-Os	0.636	0.497	0.398	0.273	0.301	0.229

(Ver.111 Ver.100 and Ver.098 stand for OpenSSL Version 1.1.1, Version 1.0.0 and Version 0.9.8)

Another limitation is that the detection rate might be low in library-like binaries (.so files, etc.). This is unavoidable because functions in library binaries tend to be independent without caller-callee relationships to recover. But we wont regard this as a critical problem, because the original intention of RouAlign is an auxiliary tool to help researchers trace out the calling routine from a specified function node.

6 Related Works.

Function alignment by FCGs is a long proposed topic, but poorly studied. [17] is about the first to introduce the Hungarian algorithm into binary analysis, e.g., malware classification. Further studies [25] use FCG merely on binary similarity discrimination rather than more detail analysis. Bindiff [9] introduces a MD index [8] to represent the topologic of function in FCG. But the MD index only counts the 1st order features like in-out degrees and could perform "medium" good in practice [10]. Recently, αdiff [20] uses FCG with DNN to detect cross-version binary code similarity and achieves some better results than Bindiff.

Cross-version function recognition is a hot topic recently. Dynamic analyzing methods assume that similar functions perform similar runtime behaviors. For example, Bingo [6], etc., capture behaviors of a function with various contexts. However, coverage and contexts generating are still problems for dynamic methods. Static methods utilize instructions and raw bytes to calculate the similarity between functions. They are not good at cross-version scenarios, and many researchers are trying to resolve this problem. For cross-platform problems, Gemini etc. [15,21], extract structure features and basic block features to calculate the similarity. For cross-optimization problems, Asm2vec [7] and InnerEye [26] map instructions and opcodes into high dimensional vector space, and then value the similarity by these mathematical representations.

7 Conclusion

In this paper, starting from the requirement of recognizing a calling routine from cross-version binaries, we proposed a novel method to learn high-level features of function call graphs to recover the caller-callee relationships between functions. We design a model to align routine and functions called RouAlign and series experiments to compare RouAlign with popular tools in real world. The evaluation results show that RouAlign outperforms the widely used commercial tools by over 35 percentages on average precision. We successfully reveal the great potential of function call graphs in function recognition and our Graphlet Edge Embedding method indicates a possible direction in the future.

Acknowledgments

We thank anonymous reviewers for their invaluable comments and suggestions. Can Yang and Jian Liu share the co-first authorship.

References

1. Song, Le. ”Structure2Vec: Deep Learning for Security Analytics over Graphs”. 2018.
2. Bromley, Jane, et al. ”Signature verification using a” siamese” time delay neural network.” Advances in neural information processing systems. 1994.
3. Altschul, et al. ”Basic local alignment search tool.” Journal of molecular biology 215.3 (1990): 403-410.
4. Shi, Chuan, et al. ”A survey of heterogeneous information network analysis.” IEEE Transactions on Knowledge and Data Engineering 29.1 (2016): 17-37.
5. Andriesse, Dennis, et al. ”An in-depth analysis of disassembly on full-scale x86/x64 binaries.” 25th USENIX Security Symposium (USENIX Security 16). 2016.
6. Chandramohan, Mahinthan, et al. ”Bingo: Cross-architecture cross-os binary search.” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016.
7. Ding, Steven, et al. ”Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization.”. IEEE, 2019.
8. Dullien, et al. Automated attacker correlation for malicious code. BOCHUM UNIV (GERMANY FR), 2010.
9. Dullien, Thomas, and Rolf Rolles. ”Graph-based comparison of executable objects (english version).” SSTIC 5.1 (2005): 3.
10. BinDiff manual. `https://www.zynamics.com/bindiff/manual/`. Last accessed 15, Sept 2019.
11. Pascal Junod, et al.”Obfuscator-llvm–software protection for the masses.” In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9. IEEE, 2015.
12. Eschweiler, Sebastian, Khaled Yakdan, and Elmar Gerhards-Padilla. ”discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code.” NDSS. 2016.

13. Feng, Muyue, et al. "Open-Source License Violations of Binary Software at Large Scale." IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019.
14. Feng, Qian, et al. "Scalable graph-based bug search for firmware images." Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2016.
15. Xu, Xiaojun, et al. "Neural network-based graph embedding for cross-platform binary code similarity detection." Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2017.
16. Grover, Aditya, and Jure Leskovec. "node2vec: Scalable feature learning for networks." Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2016.
17. Hu, Xin, et al. "Large-scale malware indexing using function-call graphs." Proceedings of the 16th ACM conference on Computer and communications security. ACM, 2009.
18. Kuchaiev, Oleksii, et al. "Topological network alignment uncovers biological function and phylogeny." Journal of the Royal Society Interface 7.50 (2010): 1341-1354.
19. Lszl, Tmea, and kos Kiss. "Obfuscating C++ programs via control flow flattening." Annales Universitatis Scientarum Budapestinensis de Rolando Etvs Nominatae, Sectio Computatorica 30.1 (2009): 3-19.
20. Liu, Bingchang, et al. "αdiff: cross-version binary code similarity detection with dnn." Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, 2018.
21. Luo Mengxia, Can Yang, and Yu Lei. "Funcnet: A euclidean embedding approach for lightweight binary association analysis." EAI International Conference on Security and Privacy in Communication Networks 2019: in press.
22. Milenkovi, Tijana, and Nataa Prulj. "Uncovering biological network function via graphlet degree signatures." Cancer informatics 6 (2008): CIN-S680.
23. Milo, Ron, et al. "Network motifs: simple building blocks of complex networks." Science 298.5594 (2002): 824-827.
24. Tang, Jian, et al. "Line: Large-scale information network embedding." Proceedings of the 24th international conference on world wide web. International World Wide Web Conferences Steering Committee, 2015.
25. Tang, Yong, et al. "Matching Function-Call Graph of Binary Codes and Its Applications (Short Paper)." International Conference on Information Security Practice and Experience. Springer, Cham, 2017.
26. Zuo, Li, et al. Neural machine translation inspired binary code similarity comparison beyond function pairs. In *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019. in press
27. SimHash wiki. `https://en.wikipedia.org/wiki/SimHash`. Last accessed 3rd, Jan. 2020.