BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables

Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, Aiman Hanna

BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables

Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna

Concordia University, Montreal, Canada

Abstract. Binary code fingerprinting is a challenging problem that requires an in-depth analysis of binary components for deriving identifiable signatures. Fingerprints are useful in automating reverse engineering tasks including clone detection, library identification, authorship attribution, cyber forensics, patch analysis, malware clustering, binary auditing, etc. In this paper, we present BinSign, a binary function fingerprinting framework. The main objective of BinSign is providing an accurate and scalable solution to binary code fingerprinting by computing and matching structural and syntactic code profiles for disassemblies. We describe our methodology and evaluate its performance in several use cases, including function reuse, malware analysis, and indexing scalability. Additionally, we emphasize the scalability aspect of BinSign. We perform experiments on a database of 6 million functions. The indexing process requires an average time of 0.0072 seconds per function. We find that BinSign achieves higher accuracy compared to existing tools.

Keywords: Code Fingerprinting; Static Analysis; Reverse Engineering

1 Introduction

1.1 Reverse Engineering and Function Fingerprinting

Fingerprinting binary functions can be of paramount importance in reverse engineering. Function fingerprinting has many applications including compiler identification [3], authorship analysis, clone detection, vulnerability detection, provenance analysis, malware detection, malware classification [2], etc. One benefit of function fingerprinting is tagging a suspicious binary as malicious or benign. The number and complexity of malware attacks have been growing significantly. In 2015, around 431 million new malware variants were uncovered [6]. Contrary to conventional signature-based detection, heuristic-based techniques are more effective and robust in detection and classification of new variants [35].

1.2 Approach

An effective fingerprinting approach produces a unique and compact representation of the functionality of a binary function. Functions that perform similar functionalities should be assigned similar fingerprints. It must also be robust to

minor byte-level discrepancies and differences in the structure of the Control-Flow-Graph (CFG). The probability of a fingerprint collision must be negligible. In this paper, we present BinSign, a fingerprinting framework for binary functions, which consists of two main components: (1) Scalable fingerprint generation and indexing of a large dataset, (2) Fingerprint matching. Our fingerprint generation approach relies on a set of features that are extracted from assembly functions combined with structural information from partial CFG traces.

To avoid pairwise comparison of a large volume of fingerprints, we use three mechanisms to facilitate achieving a scalable matching process. First, we leverage Locality-Sensitive Hashing (LSH) [34] and min-hashing [11] for selecting candidates. Second, we apply a filter based on the number of basic blocks of the CFG. Finally, BinSign system is distributed on multiple machines using Rabbit MQ [32] in order to further improve the performance and scalability. In addition, BinSign utilizes the Jaccard similarity to compute similarity scores between the target function and each candidate. To improve accuracy, features are ranked according to their significance while calculating the similarity score.

1.3 Contributions

Our contributions can be summarized as follows:

- We propose a fingerprinting approach for binary functions using features that capture syntactic, semantic and structural information of a function.
- We design and implement an efficient and scalable matching framework to match a target function fingerprint against a large repository of fingerprints.
- We evaluate BinSign in several use cases, including function reuse, malware analysis, obfuscation resilience, and function indexing scalability.

1.4 Paper Organization

This paper is structured as follows. Section 2 presents the underlying algorithmics of the components of our fingerprinting methodology, including the details of the fingerprint generation and matching algorithms. Experimental results are discussed in Section 3. Section 4 reviews the state-of-the-art approaches. Concluding remarks are ultimately presented in Section 5.

2 BinSign Methodology

BinSign utilizes meaningful features for function fingerprinting, which comprises of two main steps: fingerprint generation and fingerprint matching.

2.1 Threat Model

We design BinSign to help in the reverse engineering process. It is designed to be resilient to changes in the code such as those introduced by different compilers. The fingerprints are designed to be resilient to light obfuscation including register

replacement, register reassignment, dead-code insertion, code substitution, name stripping, and removal of symbolic information. BinSign is not intended to replace the reverse engineering process, but merely to support it. Unpacking and de-obfuscation lie outside the scope of our threat model.

2.2 Feature Extraction

It is important to select the right features that characterize the semantics of programs. The semantics of code operations can be captured by analyzing mnemonic groups and operand types even if the symbols are stripped. Each function's fingerprint includes a feature vector $\boldsymbol{v}_{f_i} \in \boldsymbol{V}$ in the form of key/value pairs (k_i, v_i).

Table 1: Tracelet Features

Data Constants	`Constants, Strings, #Constants, #Strings`
Functionality Tags	`#API Tags, #Library Tags, #Mnemonic Groups`
Tracelet Info.	`#Instructions, #Operands, Code Refs., #Code Refs., Function Calls, #Function Calls, Imported Functions, #Imported Functions`

Table 2: Global Features

Data Constants	`#Constants, #Strings`
Prototypes	`Return Type, Arguments, #Arguments, *Arguments`
Functionality Tags	`#API Tags, #Library Tags, #Mnemonic Groups`
Function Info.	`*Function, *Local Variables, Function Flags, Tracelets, #Tracelets, #Instructions, #Code Refs., #Out Calls, #Basic Blocks`

Features are extracted at two levels: global and tracelet features. Features that describe each basic block (such as the constants in that block) are combined into tracelet features. The structure of the CFG is captured in the function's fingerprint through the tracelet features (Table 1). The global features occur once per function and describe the function as a whole, such as the return type and function size (Table 2). The symbols "#" and "$*$" denote "number of" and "size of", respectively. Some features are common to both tracelet features and global features. We take into consideration the following groups of information.
Characterization of Function Prototype: Each prototype carries valuable information, such as the return type, the number and types of arguments.
Composition of CFG Instructions: These features capture the number of basic blocks and types of instructions in each block. The mnemonics and operands

are extracted and normalized. The normalized mnemonics list contains a generalized representation of the instructions, with operands numbered according to their types. General registers are replaced with 1, memory references are coded as 2, etc. After that, a simple frequency analysis is performed: the number of occurrences of each instruction, and the total number of calls to registers and memory addresses are determined. Then, the instruction mnemonics are classified into 15 groups based on the type of operation. Next, the total number of instructions and the number of instructions per group are computed.

Types of Local, System, and API Calls: Functions can be categorized according to the execution outcome on a system. System calls are interaction points with the operating system and provide valuable information on the runtime behavior. These calls are used to assign functionality tags. A functionality tag is an annotation assigned to code fragments that provides a high-level description of the context and side effects. Tags are useful for fast identification of specific groups of operations. As described in [33], a function is assigned multiple tags if it encloses several system calls. Moreover, combinations of functionality tags can describe the overall functionality of the code regions and highlight the sequence of actions carried out for performing them (e.g., CRY+FIL+NET is translated to crypto-operations on a file, followed by network communication).

2.3 Fingerprint Generation

The fingerprint generation process is depicted in Figure 1. The approach consists of: (1) Disassembling and CFG extraction, (2) tracelet generation and feature extraction, and (3) feature hashing. In the following, we detail these steps.

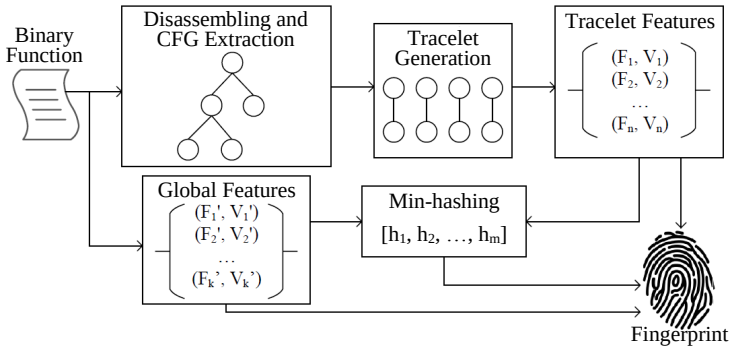


Fig. 1: Fingerprint Generation

Disassembling and CFG Extraction: Given a binary file, the first step is to disassemble it using the industry-standard disassembler *IDA Pro* [8]. In order to capture the structural information of a function, we take into consideration its CFG because it captures syntactic elements (assembly instructions) and relationships (jumps/calls) between blocks.

Tracelet Generation: One of the objectives of BinSign is to generate fingerprints that capture not only the syntactic information of a function, but also its structure. The intent is to capture all execution traces of a function. However, extracting all paths from a CFG is computationally expensive, especially for large functions. Moreover, it would be redundant to consider the common nodes between different paths multiple times. To counter this issue, we adopt the idea presented in [17], by decomposing function CFGs into partial traces of execution, namely tracelets. In BinSign, tracelets are comprised of two basic blocks. This means that each edge in the CFG results in one tracelet. Considering a larger number of basic blocks per tracelet results in redundant information without offering additional benefits. This solution allows us to generate execution traces more efficiently and without losing in functionality since all CFG nodes and edges are visited. After disassembling the function, extracting its CFG, and generating tracelets, the features described in section 2.2 are extracted.

Signature Hashing: We apply *min-hashing* [11] in order to produce a compact representation of the signature. In essence, min-hashing is a technique that reduces the dimensionality of a set using a number of hash functions. Min-hashing is applied to the normalized instructions. The function's normalized instructions constitute a suitable representation of the functionality and can be used effectively in the fingerprint matching process. Each instruction is normalized by replacing the operands with a number that represents the operand's type. Then, every normalized instruction is hashed using all hash functions and the minimum hash value is selected. The collection of the minimum hash values represents the signature hash. Multiple random numbers (seeds) are used in order to generate many different hash functions. The seeds are generated randomly beforehand, and remain constant every time a new fingerprint is generated.

2.4 Fingerprint Matching

Through the matching process (Figure 2), a candidate set is selected. After that, the similarity between each candidate and the target is computed.

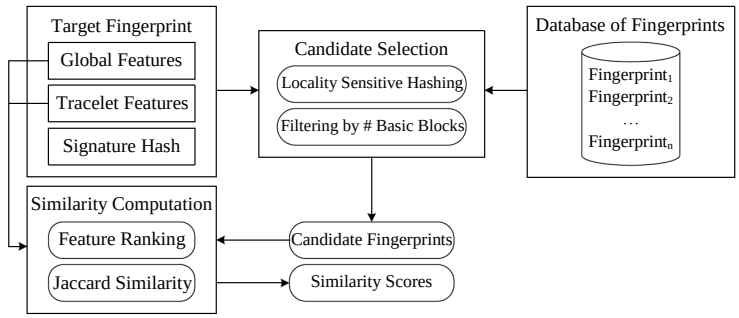


Fig. 2: Fingerprint Matching

Fingerprint Candidate Selection: Two filters are implemented. First, the functions are filtered by the number of basic blocks. Second, LSH [34] is used. The idea is to divide the hash values of the min-hash signature into bands. Each band consists of multiple hash values. A fingerprint from the dataset is considered as a candidate if all hash values match in at least one band. The values of each band in the target min-hash signature is used to create a query to obtain candidates from the database. The results of all the queries are combined into the candidate set. Through multiple experiments, we find that a band size of *seven* hash values and a total of *thirty* bands constitute a suitable signature. Since LSH approximates the Jaccard similarity, this choice of band size and number of bands sets the similarity threshold to about 60%. According to [34] an approximation of the threshold is calculated as $(\frac{1}{b})^{\frac{1}{r}}$, where b represents the number of bands and r the size of band. To further reduce false positives, we also filter the fingerprints based on the number of basic blocks. It is unlikely for a function with a small number of basic blocks to match a function with a significantly larger number of blocks. Through experiments, we find that a threshold of 30% for the difference in the number of basic blocks is appropriate.

Fingerprint Similarity Computation: A similarity score between the target fingerprint and each fingerprint in the candidate set is calculated using the Jaccard similarity. More precisely, the Jaccard similarity between the global features is calculated then combined with the Jaccard similarity between the tracelet features. Each feature in the fingerprint has a different effect on the similarity score, as some features are more significant than others. Therefore, each feature that influences the similarity score has a different weight. The weights are assigned using the gain ratio attribute evaluation algorithm provided by Weka [10] for ranking the features. Weka is a tool that offers implementations of several machine learning algorithms. This is performed through a supervised machine learning process where the names of the functions are known.

3 Experimental Results

We perform several experiments to evaluate BinSign in terms of accuracy, performance, and scalability. We also compare the accuracy with existing tools.

3.1 Dataset Description

To evaluate the scalability, we include fingerprints of 6 million functions generated from well-known libraries, malware samples, and system dynamic library files from `Microsoft Windows`. Additionally, we focus our matching experiments on slightly more than 23,000 functions of our dataset for a more precise evaluation of the accuracy. These functions are from different versions of the libraries: `libpng`, `sqlite`, and `zlib`. These files are compiled using `Visual Studio` (`MSVC`) 2010 and 2013. `Zeus` and `Citadel` malware samples are also included in the dataset. In our experiments, function names are not used during the matching

process, but only for verification. If the candidate with the highest similarity score does not have the same name as the target function, we examine both functions manually. Manual examination is only performed for verification to calculate the accuracy, but it is not required by BinSign.

3.2 Comparison with Existing Tools

This experiment compares the accuracy of BinSign against `Diaphora` [7] and `PatchDiff2` [9]. The tools `Diaphora` and `PatchDiff2` are both IDA Pro plugins that can be used for comparing binary files. `Diaphora` offers different options when matching binary functions. We deactivate the option of using unreliable methods. We also activate the option of ignoring the function names.

We use the libraries `libpng`, `sqlite`, and `zlib` compiled using two compilers. We then compare the two binary files of each library resulting from the compilation using `MSVC` 2010 and `MSVC` 2013. By using two different compilers, we introduce some noise into the binary functions so that there are some differences introduced by the compilers. We attempt to match the functions in the file compiled by `MSVC` 2010 as the target set against the functions in the file compiled by `MSVC` 2013. We match each function in the target set using BinSign by finding the function in the candidate set with the highest similarity score.

Table 3: Function Matching Comparison Between Tools

Tool Name	Library	#Target Functions	#Correct Matches	Accuracy
Diaphora	libpng	620	408	65.81%
	sqlite	1489	657	44.12%
	zlib	156	79	50.64%
PatchDiff2	libpng	620	510	82.26%
	sqlite	1489	937	62.92%
	zlib	156	122	78.21%
BinSign	libpng	620	553	89.19%
	sqlite	1489	1391	93.42%
	zlib	156	134	85.90%

Table 3 displays the results of the comparison. The accuracy is calculated by finding the percentage of the correctly matched functions to the total number of functions in the binary file. BinSign consistently achieves the highest accuracy between the tools being compared. This is due to the fuzziness of the method BinSign is using to perform the matching. This allows BinSign to be more lenient when dealing with the modifications presented by different compilers. The difference in accuracy is due to the types of features considered by these tools and that they require an exact match for some of the features.

3.3 Function Reuse Detection

We attempt to detect reused functions between versions 1.2.5, 1.2.6, 1.2.7, and 1.2.8 of the `zlib` library, as well as between the libraries `zlib` and `libpng`.

Each version is used to match the corresponding functions in its consecutive version. We also attempt to match reused functions from zlib library in libpng. After manual inspection, we identify 52 reused functions. The function is considered to be matched correctly if the corresponding function in the other version of the library is ranked first with the highest similarity score.

Table 4: Function Reuse Detection Results

Library Versions	#Functions	Threshold	Accuracy	#Candidates	Time
zlib1.2.5-zlib1.2.6	169	60%	89.47%	24970	3.9s
		65%	78.11%	11372	2.8s
zlib1.2.6-zlib1.2.7	183	60%	94.67%	30881	3.7s
		65%	90.71%	14947	2.5s
zlib1.2.7-zlib1.2.8	178	60%	98.79%	26965	4.5s
		65%	88.76%	12003	3.7s
zlib1.2.8-libpng1.6.17	52	60%	100%	4474	4.7s
		65%	100%	2147	4.4s

The results are shown in Table 4. The LSH similarity threshold is set once to 60% and once to 65%. The number of candidates presented in Table 4 is the sum of the size of the candidate sets for all target functions. Figure 3 plots the number of target functions against the total number of candidates. Increasing the number of functions increases the total number of candidates. There are other factors that affect the total number of candidates such as the size of the target functions. Smaller functions contain fewer distinctive features. As a result, they tend to have a larger candidate set. Therefore, the total number of candidates does not only depend on the number of target functions, but also on their characteristics and size. Figure 4 shows how the number of basic blocks of the functions affects the matching time. Most of the time is spent on computing the similarity scores. Therefore, the number of candidates and the size of the functions are important factors that affect the matching time.

The difference in accuracy when the LSH threshold is set to 60% against 65% is more significant in the first and third rows of Table 4. This is due to more functions with LSH similarity scores lying between 60% and 65% in these cases. When we compare the second and third rows of the table, we see that the third row has a better accuracy score when the threshold is set to 60%. However, this is not the case when it is set to 65%. This is because the comparison displayed in the third row includes more functions with similarity scores between 60% and 65%, which are filtered out when the threshold is increased.

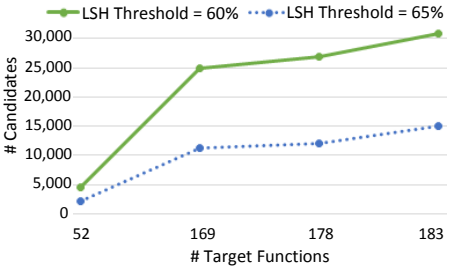


Fig. 3: Number of Target Functions vs. Number of Candidates

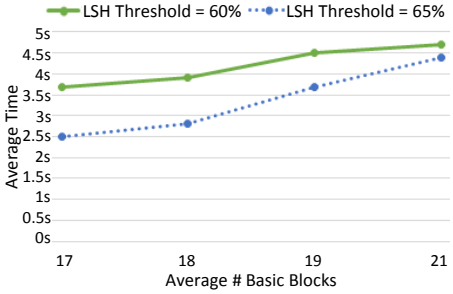


Fig. 4: Number of Basic Blocks vs. Matching Time

3.4 Scalability Evaluation

We index 6 million fingerprints and measure the time it takes. On average, the indexing process requires around 0.0072 seconds per function. This includes the time for fingerprint generation and database communication.

Fingerprint Methodology Scalability: Using min-hashing and LSH, we enhance the scalability by selecting a candidate set through the banding technique. To speed up the process, only candidates that are selected through our filters are considered instead of brute force matching.

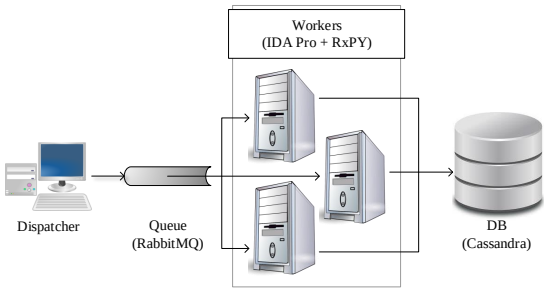


Fig. 5: Architecture of the Distribution Process

Implementation Scalability: We implement a distribution mechanism (Figure 5) using RabbitMQ [32]. It is an open source messaging software based on the international standard Advanced Message Queuing Protocol (AMQP) [5]. Thanks to its simplicity, we find that it is more suitable to our purposes than other distribution frameworks that require a lot of processing for data analysis and synchronization with the server, leading to a lot of overhead, and thus slowing the process. To index several files, the binaries are distributed to different workers. The distribution is done depending on the power of each worker machine. Each worker runs multiple instances of IDA Pro simultaneously to process

the files, generate the fingerprints, and store them in the database. Note that RxPY (reactive extension) is used to run the code in an asynchronous manner. The distribution is performed on a server machine with an Intel Xeon CPU E5-2630 v3 @2.40 GHz (2 processors) and 128 GB of RAM running Microsoft Windows Server 2008 64-bit, along with a PC with an Intel Core i7 CPU 920 @2.67 GHz and 12 GB of RAM running Microsoft Windows 7 64-bit.

3.5 Resilience to Different Compiler Optimization Levels

We use `MSVC` 2013 to compile version 1.2.8 of `zlib` with four optimization levels. `MSVC` 2013 offers the optimization levels: disabled optimization (Od), minimize size (O1), maximize speed (O2), or full optimization (Ox). We compile the `zlib` library with full optimization (Ox) and use the resulting binary functions as the target to match functions compiled using other optimization levels.

The results are presented in Table 5. The lowest accuracy score of 65.05% occurs when matching the fully optimized (Ox) functions against disabled optimization (Od). Compiling with optimization level O1 produces more similar functions to the fully optimized functions, which results in 87.85% accuracy. We find that compiling with the optimization levels Ox and O2 seem to produce identical assembly code in this case. Therefore, comparing the code compiled with O2 against O1 would produce similar results to Ox against O1.

Table 5: Results of Matching Different Optimization Levels

Optimization Levels	Overall Accuracy	Average Time
`Ox vs. Od`	`65.05%`	`4.2s`
`Ox vs. O1`	`87.85%`	`4.3s`
`Ox vs. O2`	`100.00%`	`0.26s`

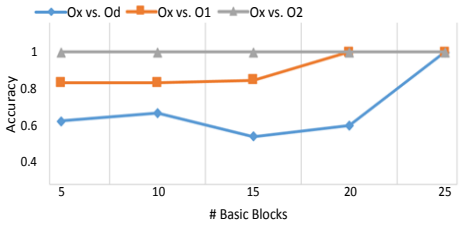


Fig. 6: Number of Basic Blocks vs. Accuracy

After taking a closer look, we find that the size of the target functions has an effect on the accuracy. The accuracy of matching functions with different number of basic blocks is displayed in Figure 6. The functions with higher number of basic blocks are matched with higher accuracy. This is due to the fact that larger functions tend to contain more distinctive features. Small functions usually have features and structures that are fairly common. Therefore, there is a higher probability of mismatching smaller functions than larger ones.

When comparing the optimization levels Ox and Od, we notice a dip in the graph, which might seem counter-intuitive. This is because smaller functions (with 5 to 10 basic blocks) may have less room for optimization. However, larger functions (with 10 to 20 basic blocks) contain higher possibilities for optimization. Functions that are even larger (with around 25 basic blocks) tend to have more distinctive features, which increases the accuracy of the matching process.

3.6 Malware Similarity Analysis

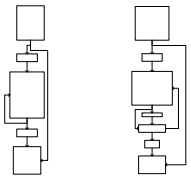
Since the functions in Citadel are derived from the functions in Zeus [30], we match one of these functions, namely the RC4 function. Citadel reuses the RC4 stream cipher function from Zeus with minor modifications [33]. IDA Pro identifies 642 functions in Zeus and 896 functions in Citadel. We then generate fingerprints for all the functions and attempt to match the RC4 function. The time it takes to match the RC4 function is 0.463 seconds.

We find that the top match was in fact the modified RC4 function with a similarity score of 0.68787. The other matches with lower similarity scores are from different library files. The CFG of RC4 functions in Zeus and Citadel can be seen in Figure 7a and Figure 7b respectively. Although the number of basic blocks in the two CFG's is different, the RC4 function is still identified because the difference in the number of basic blocks is less than 30%.

Since many of the functions in Citadel are derived from the functions in Zeus, we attempt to match all the functions in Zeus. BinSign matches 591 out of 642 functions in Zeus to functions in Citadel. Out of the matched functions, 517 functions are matched with a high similarity score above 90%.

Table 6: Candidates of RC4 Function from Citadel

Function	Similarity Score
sub_42E92D	0.68787
sub_10034D0A	0.40042
png_set_sCAL	0.35377



(a) (b)

Fig. 7: CFG of RC4 Function in (a) Zeus and (b) Citadel

3.7 Obfuscation Resilience

Although heavy obfuscation is out of the scope of our threat model, we provide some insight into its effects. We use the Obfuscator-LLVM [25] for this purpose. It offers three obfuscation techniques: control flow flattening, instruction substitution, and bogus control flow. After applying the three obfuscation techniques to a piece of code in C++, we add the three different obfuscated functions to the dataset. The function that results from instruction substitution is identified as a match with a high similarity score of 0.84399, which shows resilience to this type of obfuscation. However, the obfuscated functions resulting from control flow flattening and bogus control flow are not identified. The number of basic blocks increased by more than 30%. Therefore, the fingerprints are filtered out by our filter that considers the number of basic blocks.

4 Related Work

This section reviews research on binary fingerprinting and related domains.

4.1 Exact and Inexact Fingerprint Matching

Several studies focus on fingerprinting library functions in binary files (e.g., [12, 22,23]). Some approaches use exact matching and apply byte-level sequence patterns. *IDA Pro* is an industry-standard disassembler that is widely used by reverse engineers [23,28]. It provides a built-in capability for recognizing standard library functions and hex code sequences that are generated by common C-family compilers through *IDA F.L.I.R.T* [22]. Although *F.L.I.R.T* is a useful mechanism, it has limitations. Library functions with small byte-level discrepancies are not recognized. The inability to perform inexact matching is a major limitation. Other studies examine flexible signature matching techniques alongside graph-based analysis to measure the similarity between programs [12,13,18,23]. Such approaches show higher recall in comparison to exact matching.

4.2 Graph-Based Binary Fingerprinting

The BinDiff algorithm [18] inspired some works that use graph matching for various purposes (e.g., [4, 13–15, 21, 23, 27, 28]). One limiting factor is placing the attention on structural similarity, ignoring the instruction semantics of each basic block. Instruction hash comparison is a simple, fast technique to fingerprint functions and assembly instructions [24]. A hash value can represent the semantics of basic blocks. The work in [29] terms such values as semantic juice.

4.3 Source and Binary Clone Detection

Research on source code clone detection investigates four types of clones [16,26]. Type I clones are exact clones. Type II clones preserve the syntactic structure with changes in identifiers, layout, comments, and types. Type III clones have altered fragments. Type IV clones are semantic clones. Applying source code clone detection techniques to disassemblies is challenging due to limited source-level information. The research on binary clone detection measures the similarity between binary files [1,12,15,19,20,31]. The approach presented in [15] measures the provenance similarity of binaries using symbolic execution. Morphological analysis and static code synchronization is another solution for binary clone detection [12]. These approaches are good candidates for detecting types III and IV clones. The dynamic analysis techniques that monitor input/output behavior of functions result in lower false positive rates in detection of type IV clones [24].

5 Conclusion

In this paper, we defined the main components of function fingerprinting, described the algorithms, and the processes of fingerprint generation and matching. The methodology was evaluated in terms of function matching, malware analysis, obfuscation resilience, and scalability. We showed that the methodology is effective and can improve the accuracy of exact and inexact fingerprint matching. BinSign outperformed existing tools and achieved a higher accuracy score.

We also described different measures undertaken to ensure BinSign's scalability. BinSign performed efficient fingerprint generation of 6 million functions such that the indexing process required 0.0072 seconds per function on average.

Acknowledgments. This work is supported by a research grant under the Department of National Defence/Natural Sciences and Engineering Research Council of Canada (NSERC) in collaboration with Google. In addition, the authors acknowledge the contribution of Nhat Nguyen in assisting with the implementation of the distribution mechanism and the web interface.

References

1. Huang, H., Youssef, A., Debbabi, M.: BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. Accepted for publication in the ACM Asia Conference on Computer and Communications Security (ASIACCS). ACM Press (2017).
2. Alrabaee, S., Shirani, P., Debbabi, M., Lingyu, W.: On the Feasibility of Malware Authorship Attrubution. In: International Symposium on Foundations and Practice of Security, pp. 256–272. Springer, Cham (2016)
3. Rahimian, A., Shirani, P., Alrabaee, S., Lingyu, W., Debbabi, M.: BinComp: A Stratified Approach to Compiler Provenance Attribution. Digital Investigation. 14, S146–S155 (2015)
4. Alrabaee, S., Shirani, P., Lingyu, W., Debbabi, M.: Sigma: A Semantic Integrated Graph Matching Approach for Identifying Reused Functions in Binary Code. Digital Investigation. 12, S61–S71 (2015)
5. Advanced Message Queuing Protocol (AMQP), `https://www.amqp.org/`
6. Internet Security Threat Report 2016, `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf`
7. Diaphora: A Program Diffing Plugin for IDA Pro, `https://github.com/joxeankoret/diaphora`
8. Hex-Rays IDA Pro, `https://www.hex-rays.com/products/ida/`
9. Diffng Plugin for IDA, `https://code.google.com/p/patchdiff2/`
10. Weka: Machine Learning Software, `https://weka.wikispaces.com/`
11. Andoni, A., Indyk, P.: Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. Commun. ACM. 51, 117–122 (2008)
12. Bonfante, G., Marion, J., Sabatier, F., Thierry, A.: Code Synchronization by Morphological Analysis. Malicious and Unwanted Software. 7, 112–119 (2012)
13. Bourquin, M., King, A., Robbins, E.: Binslayer: Accurate Comparison of Binary Executables. In Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. 13, 4:1–4:10 (2013)
14. Cesare, S., Xiang, Y., Zhou, W.: Control Flow-Based Malware Variant Detection. Dependable and Secure Computing, IEEE Transactions. 11, 307–317 (2014)
15. Chaki, S., Cohen, C., Gurfinkel, A.: Supervised Learning for Provenance-Similarity of Binaries. In Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '11, 15–23 (2011)
16. Cordy, J.R., Roy, C.K.: Efficient Checking for Open Source Code Clones in Software Systems. In: 19th IEEE ICPC, pp. 217–218 (2011)
17. David, Y., Yahav, E.: Tracelet-Based Code Search in Executables. In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14, 349–360 (2014)

18. Dullien, T., Rolles, R.: Graph-Based Comparison of Executable Objects (english version). SSTIC. 5, 1–3 (2005)
19. Elva, R., Leavens, G.: Semantic Clone Detection Using Method IOE-Behavior. In: 6th International Workshop on Software Clones (IWSC), pp. 80–81 (2012)
20. Farhadi, M.R., Fung, B.C.M., Charland, P., Debbabi, M.: Binclone: Detecting Code Clones in Malware. In: International Conference on Software Security and Reliability. 8, 78–87 (2014)
21. Gascon, H., Yamaguchi, F., Arp, D., Rieck, K.: Structural Detection of Android Malware Using Embedded Call Graphs. In: Proceedings of the ACM Workshop on Artificial Intelligence and Security. AISec '13, 45–54 (2013)
22. Hex-Rays. Fast Library Identifiation and Recognition Technology: In-Depth, `https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml`
23. Jacobson, E.R., Rosenblum, N., Miller, B.P.: Labeling Library Functions in Stripped Binaries. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools. PASTE '11, 1–8 (2011)
24. Jin, W., Chaki, S., Cohen, C., Gurfinkel, A., Havrilla, J., Hines, C., Narasimhan, P.: Binary Function Clustering Using Semantic Hashes. In: 11th International Conference on Machine Learning and Applications (ICMLA). 1, 386–391 (2012)
25. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM: Software Protection for the Masses. In: B. Wyseur, editor, Proceedings of the IEEE/ACM 1st International Workshop on Software Protection. SPRO'15, 3–9 (2015)
26. Keivanloo, I., Rilling, J., Charland, P.: Internet-Scale Real-Time Code Clone Search via Multi-Level Indexing. In: 18th Working Conference on Reverse Engineering (WCRE), pp. 23–27 (2011)
27. Khoo, W.M., Mycroft, A., Anderson, R.: Rendezvous: A Search Engine for Binary Code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. MSR '13, 329–338 (2013)
28. Kinable, J., Kostakis, O.: Malware Classification Based on Call Graph Clustering. Journal in Computer Virology. 7, 233–245 (2011)
29. Lakhotia, A., Preda, M.D., Giacobazzi, R.: Fast Location of Similar Code Fragments Using Semantic 'Juice'. In: Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop. PPREW '13, 5:1–5:6 (2013)
30. Milletary, J.: Citadel Trojan Malware Analysis. Technical report, Dell SecureWorks Counter Threat Unit Intelligence Services (2012)
31. Ng, B.H., Prakash, A.: Expose: Discovering Potential Binary Code Re-use. In: COMPSAC, IEEE 37th Annual, pp. 492–501 (2013)
32. Pivotal Software. RabbitMQ Web Site, `https://www.rabbitmq.com/`
33. Rahimian, A., Ziarati, R., Preda, S., Debbabi, M.: On the Reverse Engineering of the Citadel Botnet. In: Danger, J.L., Debbabi, M., Marion, J.Y., Garcia-Alfaro, J., Zincir Heywood, N., editors, Foundations and Practice of Security (LNCS), pp. 408–425 (2014)
34. Rajaraman, A., Ullman, J.D.: Mining of Massive Datasets. Cambridge University Press (2014)
35. Ye, Y., Wang, D., Li, T. Jiang, Q.: An Intelligent Pe-malware Detection System Based on Association Mining. Journal in Computer Virology, 4, 323–334 (2008)