BinShape: Scalable and Robust Binary Library Function Identification Using Diverse Features

Paria Shirani, Lingyu Wang, and Mourad Debbabi

Concordia University, Montreal, Canada

Abstract. Identifying library functions in program binaries is important to many security applications, such as threat analysis, digital forensics, software infringement, and malware detection. Today's program binaries normally contain a significant amount of third-party library functions taken from standard libraries or free open-source software packages. The ability to automatically identify such library functions not only enhances the quality and the efficiency of threat analysis and reverse engineering tasks, but also improves their accuracy by avoiding false correlations between irrelevant code bases. Existing methods are found to either lack efficiency or are not robust enough to identify different versions of the same library function caused by the use of different compilers, different compilation settings, or obfuscation techniques. To address these limitations, we present a scalable and robust system called *BinShape* to identify standard library functions in binaries. The key idea of *BinShape* is twofold. First, we derive a robust signature for each library function based on heterogeneous features covering CFGs, instruction-level characteristics, statistical characteristics, and function-call graphs. Second, we design a novel data structure to store such signatures and facilitate efficient matching against a target function. We evaluate *BinShape* on a diverse set of C/C++ binaries, compiled with GCC and Visual Studio compilers on x86-x64 CPU architectures, at optimization levels $O0-O3$. Our experiments show that *BinShape* is able to identify library functions in real binaries both efficiently and accurately, with an average accuracy of 89% and taking about 0.14 seconds to identify one function out of three million candidates. We also show that *BinShape* is robust enough when the code is subjected to different compilers, slight modification, or some obfuscation techniques.

Keywords: Software fingerprinting; binary analysis; function identification; malware analysis

1 Introduction

Binary code analysis is an essential security capability with extensive applications, ranging from threat analysis, reverse engineering, cyber forensics, recognizing copyright infringement, to malware analysis. However, today's reverse engineers still largely rely on manual analysis with only limited support from automated tools, such as IDA Pro [7]. Such manual analysis is typically tedious and error-prone due to the complex code transformation performed by the compilers, which usually involves highly optimized control flows, varying registers, and the assignment of memory locations based on the CPU architectures and optimization settings [17]. Therefore, automated tools are highly desirable to

assist reverse engineers in binary code analysis, which is especially relevant to security applications where the source code is unavailable.

Since modern software typically contain a significant number of library functions, identifying such functions in a binary file can offer a vital help to threat analysts and reverse engineers in many practical security applications. Further, it has a strong positive impact in various applications such as clone detection [12,18], authorship attribution [45], vulnerability analysis [10,11,17,40,47], and malware analysis [32,34,48]. Since it helps to filter out those library functions and focus on the analysis of user functions. In addition, the labeled library functions could provide valuable insights about the functionality of program binaries. Hence, the ability to automatically identify library functions cannot only enhance the efficiency of such threat analysis and reverse engineering tasks, but also improve the accuracy.

Automating the process of accurately identifying library functions in binary programs poses the following challenges: *(i) Robustness:* the distortion of features in the binary file may be attributed to different sources arising from the platform, the compiler, or the programming language, which may change the structures, syntax, or sequences of features. Hence, it is challenging to extract robust features that would be less affected by different compilers, slight changes in the source code as well as obfuscation techniques. *(ii) Efficiency:* another challenge is to efficiently extract, index, and match features from program binaries in order to detect a given target function within a reasonable time, considering the fact that many known matching approaches imply a high complexity [36]. *(iii) Scalability:* due to the dramatic growth of software packages as well as malware binaries, threat analysts and reverse engineers deal with large numbers of binaries on a daily basis. Therefore, designing a system that could scale up millions of binary functions is an absolute necessity. Accordingly, it is important to design efficient data structures to store and match against a large number of candidate functions in a repository.

To address the library identification problem, security researchers elaborated techniques to automatically identify library functions in binaries. For instance, the widely-used IDA FLIRT [6,13] applies signature matching to patterns generated according to the first invariant byte sequence of the function. This simple method is indeed very efficient but the robustness is a major issue. It suffers from the limitation of signature collisions and might require a new signature for each new version as the result of a slight modification, since various compilers and build options usually would affect byte-level patterns. Similarly, most other existing methods, e.g., *UNSTRIP* [26], which is based on the interaction of wrapper functions with the system call interfaces and *libv* [41], which employs data flow analysis and graph isomorphism, also rely on one type of features and thus might also be easily affected by compiler families and compilation settings. Furthermore, these methods are usually not as efficient as FLIRT due to the need for complex operations, e.g., graph isomorphism testing.

In this paper, we aim to address aforementioned challenges and limitations of existing works. Specifically, we focus on following research problems. *Can*

we generate a "robust" signature for each library function to be resilient against compiler effects and obfuscation techniques? Can we rely on only those features who extraction, indexing, and matching can be performed in an efficient manner? Can we design an efficient data structure to allow a target function be matched against millions of candidate functions stored in a repository in a short time (e.g., less than a second)? The key idea of *Binshape* is twofold. First, we derive a robust signature for each library function, called the shape of function, based on heterogeneous features covering CFGs, instruction-level features, statistical features, and function-call graphs. The shape of a library function captures a collection of most segregative characteristics along one or more dimensions of the features, which is automatically determined using selection evaluators, such as mutual information-based feature ranking and decision tree learning. Second, the shapes of library functions are extracted and stored in a repository and indexed using a novel data structure based on B^+trees for efficiently matching against a target function.

The main advantages of our approach are as follows: First, by relying mostly on lightweight features and the proposed data structure, our technique is *efficient*, and outperforms other techniques that rely on time-consuming computations such as graph isomorphism. Second, incorporating different types of features significantly reduces the chance of signature collisions compared to most existing works which rely on a single type of features. Therefore, by extracting the aforementioned heterogeneous features and furthermore selecting the best features amongst them, our approach achieves a great deal of *robustness*. As demonstrated by the experimental results, there is only a slight drop in accuracy when the code is generated by different compilers or has been moderately modified. Third, our technique is general in the sense that it is not limited to a particular type of functions, e.g., the wrapper functions provided by standard system libraries [26]. Finally, testing against a large number (over a million) of functions in a repository confirms the *efficiency* and *scalability* of our system.

Contribution. In summary, our main contributions are enlisted as follows:

- **Extracting Heterogeneous Features:** To the best of our knowledge, this is the first effort in employing a diverse collection of features, including graph features, instruction-level characteristics, statistical characteristics, and function-call graphs, for library function identification.
- **Generating a Robust Signature:** The novel concept of function shape induces a single robust signature based on heterogeneous features, which allows our system to produce good accuracy even when the code is compiled with different compilers and compilation settings, and is subjected to slight modifications and some obfuscation techniques.
- **Proposing a Scalable Technique:** By designing a novel data structure and using filters to prune the search space, our system demonstrates superior performance and provides a practical framework for large scale applications with millions of indexed library functions.

2 Overview

This section illustrates our motivating example, explains the threat model, and finally provides an overview of our system.

2.1 Motivating Example

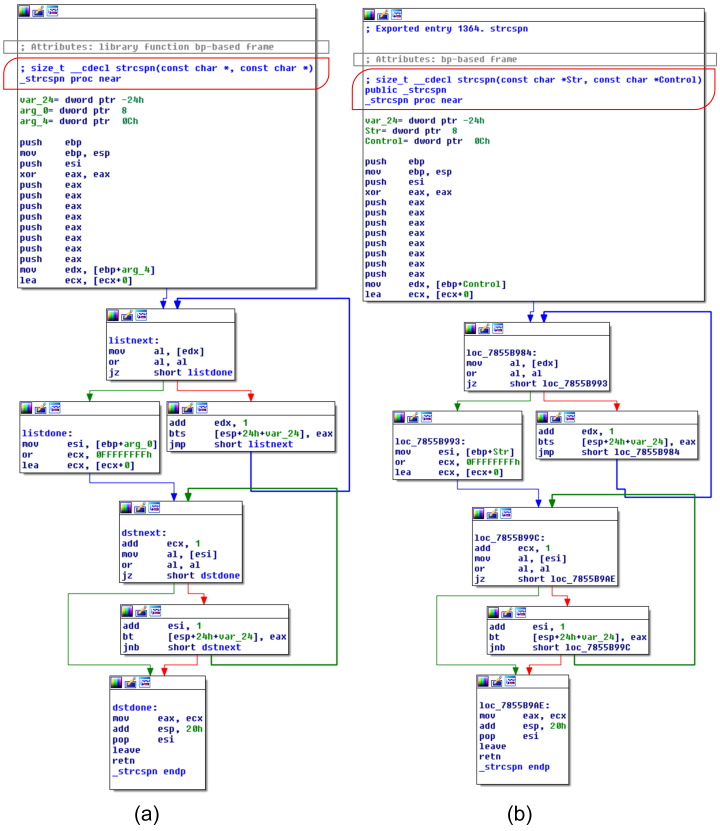

Fig. 1: CFGs of `_strcspn` function from a) a simple code and b) `msvcr90.dll`

Most of the existing works rely on a particular type of features (a review of related works is given in Section 6), and they typically organize those features as a vector. In addition, for every version of the function a new signature must be generated and indexed in the repository. Consider the CFGs and instructions of two disassembled versions of `_strcspn` function extracted from a relatively simple code and `msvcr90.dll` project, shown in Figure 1a and Figure 1b, respectively. In both functions, notice that all the instructions and CFGs of these two functions are equivalent. Indeed, the only difference appears in the function

header (surrounded by red box). As a result, FLIRT cannot identify the latter as a library function, since the first portion of the pattern generated by FLIRT is the first 32 initial byte sequence of the function [13].

Our first observation here is that, instead of using one type of features as in FLIRT, the diverse nature of library functions demands a rich collection of features in order to increase the robustness of detection. In addition, as will be demonstrated shortly in Figure 2, the most segregative features for different library functions will likely be different, and therefore a feature vector may not be the best way for representing a signature. Specifically, the CFGs of `_memmove`, `_memchr`, and `_lock_file` functions are depicted in Figure 2. We observe that two graph features of `_memmove` function are enough to be distinguishable from others in our repository. On the other hand, the CFG of `_lock_file` function contains smaller number of nodes (i.e., five), and the CFG of `_memchr` function is almost flat. Therefore, the best features to identify two different functions, one with few basic blocks and one with a large and complex CFG, would be very different; for instance, basic block level features for the former, and graph features for the latter.

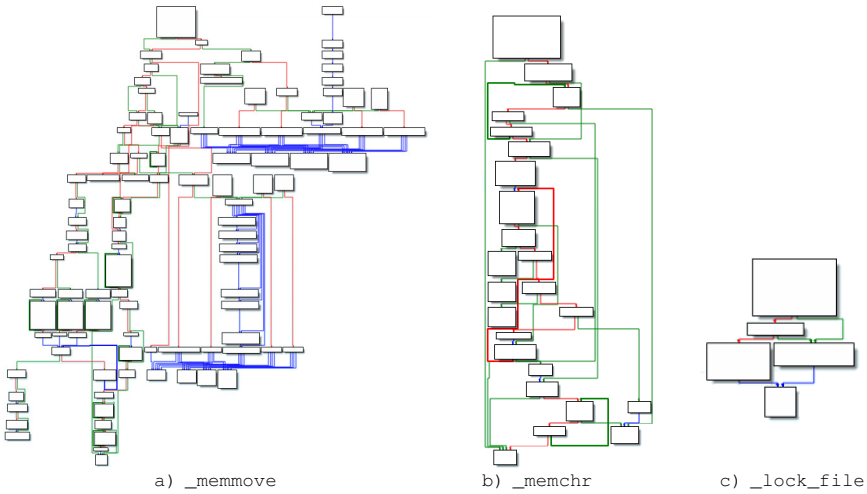

Fig. 2: CFGs of three library functions

2.2 Threat Model

In designing the features and the methodology of our system, we take into consideration several ways by which adversaries may attempt to evade detection by our system. First, adversaries may intentionally apply obfuscation techniques (discussed in more details below) to alter the syntax of binary files. Second, since

the syntax of a program binary can be significantly altered by simply changing the compilers or compilation settings, adversaries may adopt such strategies to evade detection. Finally, attackers may slightly modify the source code of library functions and reuse them so as to evade detection. However, our system is not intended to replace threat analysts or reverse engineers. Thus, it is not designed to overcome the hurdles imposed by packers, obfuscation, or encryption. Instead, our system focuses on identifying library functions in binaries that are already unpacked, de-obfuscated, and decrypted using existing tools [14, 29, 38]. Therefore, the scope of our work is limited to function identification, and our tool is designed to work together with other reverse engineering tools or efforts (e.g., those for de-obfuscation) instead of replacing them completely.

In general, obfuscation techniques may be applied to three types of binary analysis platforms, disassemblers, debuggers, and virtual machines [49]. Since we perform static analysis, the anti-disassembler category, which includes a variety of techniques such as dead code insertion, control flow obfuscation, instruction aliasing, binary code compression, and encryption will be considered in this work. We review how existing approaches handle some anti-disassemble obfuscation techniques in Section 6 (Table 7), and further evaluate *BinShape* and two other existing approaches against some obfuscation techniques through experiments in Section 5.4.

2.3 Approach Overview

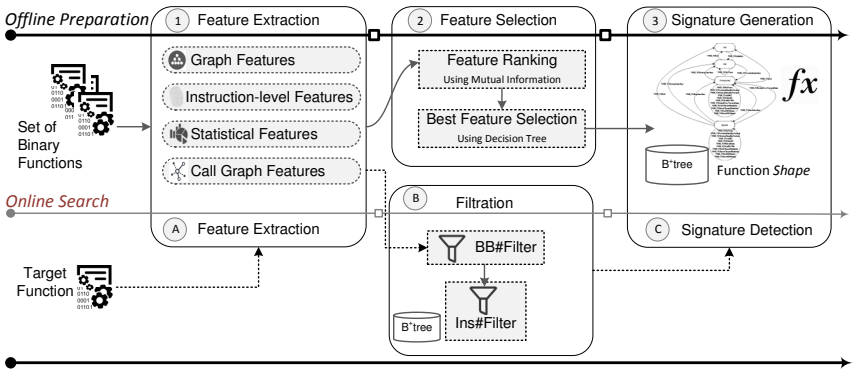

Fig. 3: Approach overview

Our approach is divided into two phases: offline preparation (indexing) and online search (detection). As illustrated in the upper part of Figure 3, the offline preparation includes: 1) feature extraction; 2) feature selection, which includes feature ranking to extract the elements of the function shape, as well as best feature selection; and 3) signature generation to index the functions in a reposi-

tory. The lower part of Figure 3 depicts online search, which includes: A) feature extraction; B) filtration; and C) detection components.

First, the binaries in our training set are disassembled by IDA Pro disassembler. Second, the graph features along with the instruction-level features, statistical features, and function-call graphs (explained in Sections 3.1, 3.2, 3.3 and 3.4 respectively) are extracted. To select subsets of the features that are useful to build the best signature, mutual information (Section 3.5) is employed on the extracted features. The top-ranked features are fed into a decision tree [20], and the outcome of the decision tree is stored in a data structure (Section 4.1) to form a signature for each library function. In addition to such signatures, we also store the top-ranked features that compose the signature of each function. For detection, all the features are extracted from a given target binary, two filters (Section 4.2) based on the number of basic blocks ($BB\#$) and the number of instructions ($Ins\#$) are used to prune the search space. Consequently, a set of candidate functions are returned as the result of filtration. Finally, the best matches are returned as the final results.

3 Feature Extraction

This section first describes different types of features, then presents feature selection, and finally defines the so-called function shape.

3.1 Graph Feature Metrics

We extract the control flow graph (CFG) of a binary function. To extract the best features for each library function and to describe the shape of a function, we define graph features based on different characteristics of the CFG. Among existing graph metrics [22], we only employ those which are inexpensive to extract. The selected graph features are listed in Table 1. Below we show an example to illustrate the application of graph features to two functions. Our graph metrics are applied to two different library functions, `memcpy_s` and `strcpy_s`, as listed in Table 1. The corresponding CFGs have identical feature values for some metrics; for instance, *numnodes*, *numedges*, and *cc* values are equal, while other metrics such as *graph_energy*, and *pearson* (shown in boldface) are different and can be used to distinguish between them in this example (more generally, we will certainly need more features to uniquely characterize a function).

As discussed before (Section 2.1), the graph features of `_memmove` function could be part of the best features, since these features can segregate `_memmove` function from others. However, graph features alone are not sufficient since there are cases where all the graph features of two different functions are identical, especially for functions of relatively small sizes. In addition, the CFG of a library function may differ due to compilation settings or slight changes in the source file. Therefore, we consider additional features in the following.

3.2 Instruction-level Features

Instruction-level features carry the syntax and semantic information of a disassembled function. Some instruction-level features are shown in Table 2a, such as

Graph Metric	Description	_memcpy_s	_strcpy_s
n, numnodes	Number of nodes	13	13
e, numedges	Number of edges	18	18
p, num_conn_comp	Number of connected components	1	1
CC	Cyclomatic complexity: $e - n + 2p$	7	7
num_conn_triples	Number of connected triples	6	6
num_loop	Number of independent loops	6	6
leaf_nodes	Number of leaves	7	7
average_degree	$2 * e/n$	2.7692	2.7692
ave_path_length	Average distance between any two nodes	**2.2308**	**2.5**
r, graph_radius	Minimum vertex eccentricity	**5**	**6**
link_density	$e/(n(n-1)/2)$	0.2308	0.2308
s_metric	Sum of products of degrees across all edges	**150**	**159**
rich_club_metric	Extent to which well-connected nodes also connect to each other	**0.2778**	**0.2778**
graph_energy	Sum of the absolute values of the real components of the eigenvalues	**18.7268**	**18.0511**
algeb_connectivity	2^{nd} smallest eigenvalue of the Laplacian	**1**	**0.3820**
pearson	Pearson coefficient for degree sequence of all edges	**0.4635**	**0.3415**
weighted_clust_coeff	Maximum value of the vector of node weighted clustering coefficients	**0.3334**	**0.5**

Table 1: Comparing graph features of *_memcpy_s* and *_strcpy_s*

the number of constants (*#constants*), and the number of callees (*#call list*). In addition, inspired by [33], we categorize the instructions according to their operation types, as shown in Table 2b. We record the frequency of each instruction category as a feature. By enriching standard CFGs with such information as different colors, there is a better chance to distinguish two functions even if they have the same CFG structure. Since these categories carry some information about the functionality of a program; for instance, encryption algorithms perform more logical and arithmetic operations.

3.3 Statistical Features

Statistical analysis of binary code can be used to capture the semantics of a function. Several works have applied opcode analysis to binary code; for instance, opcode frequencies are used to detect metamorphic malware in [42, 46]. Therefore, each set of opcodes that belong to a specific function will likely follow a specific distribution according to the functionality they implement. For this purpose, we calculate the *skewness* and *kurtosis* measures to convert these distributions into scores by the following formulas [4]:

$$Sk = \left(\frac{\sqrt{N(N-1)}}{N-1}\right)\left(\frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^3/N}{s^3}\right)$$

Feature	Description
retType	return type
declaration	declaration type
argsnum	number of arguments
argsize	size of arguments
localvarsize	size of local variables
instrnum	number of instructions
numReg	number of registers
#mnemonics	number of mnemonics
#operand	number of operands
#constants	number of constants
#strings	number of strings
#call list	number of callees

(a) Example of Instruction-level Features

Group	Description
DTR	Data transfer operations (e.g., `mov, movzx`)
STK	Stack operations (e.g., `push, pop`)
CMP	Compare operations (e.g., `cmp, test`)
ATH	Arithmetic operations (e.g., `add, sub`)
LGC	Logical operations (e.g., `and, or`)
CTL	Control transfer (e.g., `jmp, jne`)
FLG	Flag manipulation (e.g., `lahf, sahf`)
FLT	Float operations (e.g., `f2xm1, fabs`)
CaLe	System and interrupt operations (e.g., `sysexit`)

(b) Mnemonic groups

Table 2: An example of instruction-level features and mnemonic groups

$$Kz = \frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^4/N}{s^4} - 3$$

where Y_i is the frequency of each opcode, $\overline{Y}$ is the mean, s is the standard deviation, and N is the number of data points. Similarly, we calculate *Term Frequency - Inverse Document Frequency (TF-IDF)* [27] weighted opcode features, and *z-score* [50] for each opcode and for each operand, where the corpus includes all the functions in our repository.

Normalization. Each assembly instruction consists of a mnemonic and a sequence of up to three operands. The operands can be classified into three categories: memory references, registers, and constant values. We may have two fragments of a code that are both structurally and syntactically identical, but differ in terms of memory references, or registers [18]. Hence, it is essential that the assembly code be normalized prior to comparison. Therefore, the memory references and constant values are normalized to MEM and VAL, respectively. The registers can be generalized according to the various levels of normalization. The top-most level generalizes all registers regardless of types to REG. The next level differentiates General Registers (e.g., `eax`, `ebx`), Segment Registers (e.g., `cs`, `ds`), and Index and Pointer Registers (e.g., `esi`, `edi`). The third level breaks down the General Registers into three groups by size - namely, 32, 16, and 8-bit registers.

3.4 Function-Call Graph

Function-call graph is a structural representation that abstracts away instruction-level details, and can provide an approximation of a program functionality. Moreover, function-call graph is more resilient to instruction-level obfuscation that usually are employed by malware writers to evade the detection systems [21]. In addition, it offers a robust representation to detect variants of

malware programs [24]. Hence, the caller-callee relationship of the library functions is extracted.

The derived function-call graphs from those relationships are directed graphs containing a node corresponding to each function and edges representing calls from callers to callees. For labeling the nodes to exploit properties shared between functions, a neighbor hash graph kernel (NHGK) is applied to subsets of the call graph [23]. Those subsets include library functions and their neighbor functions (callees and callers).

The function G maps the features of function f_k to a bit vector of length l, where l is the number of mnemonic categories (9) as shown in Table 2. Function G checks each value of the mnemonic groups of a function; if the value is greater than 0, the corresponding bit vector is set to 1; otherwise, it would be 0.

$$G : f_k \rightarrow v_k = \{0,1\}^l$$

The neighborhood hash value h for a function f_i and its set of neighbor functions N_{f_i} can be computed using the following formula [21]:

$$h(f_i) = shr_1(G(f_i)) \oplus (\oplus_{f_j \in N_{f_i}} G(f_j))$$

where shr_1 denotes a one-bit shift right operation and $\oplus$ indicates a bit-wise XOR. The time complexity of this computation is constant time, $O(ld)$, where d is the summation of outdegrees and indegrees, and l is the length of bit vector.

For instance, suppose f_i is called by two other functions f_1 and f_2, and calls one function f_3. Therefore, the bit vectors based on the mnemonic group values of each function are generated (by the function G) to construct the set of neighbor function $N_{f_i} = \{v_1, v_2, v_3\}$. Finally, the hash value $h(f_i)$ would be equal to $(shr_1(v_i) \oplus (v_1 \oplus v_2 \oplus v_3))$.

3.5 Feature Selection

After extracting all the aforementioned features, we will end up with a number of features among which some might be the most relevant ones - those that appear more frequently and are most segregative in one function. Therefore, a feature selection process is conducted to reduce the number of features as well as to find the best ones. Our feature selection phase contains two major steps: feature ranking and best feature selection, which are described in the following.

Feature Ranking We measure the relevance of the aforementioned features based on the frequency of their appearance in each library function. Mutual information [39] represents the degree depending on which the uncertainty of knowing the value of a random variable is reduced given the value of another variable. To this end, we employ a Mutual Information (MI) measure to indicate the dependency degree between features X and library function labels Y as follows:

$$MI = \sum_{x \in X} \sum_{y \in Y} p(x,y) log_2 \frac{p(x,y)}{p(x)p(y)}$$

where x is the feature frequency, y is the class of library function (e.g., `memset`), $p(x, y)$ is the joint probability distribution function of x and y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of x and y. The main intention of this feature ranking is to shorten the training time and to reduce overfitting. We measure mutual-information-based feature ranking on all categories of aforementioned features. In addition to mutual information, we apply feature selection evaluators including *ChiSquared* [43], *GainRatio* [43], and *InfoGain* [44] using WEKA [5] on our features. Finally, we select the top-ranked features of our training dataset based on the MI.

Best Feature Selection Our aim is to build a classification system which separates library functions from non-library functions. As such, we choose to apply a decision tree classifier on the top-ranked features obtained from the feature ranking process.

Each library function is passed through the decision tree and the best provided features for that specific function are recorded. This automated task is performed on all library functions to create a signature for each. For instance, according to our dataset, the best features for `_strstr` function are *#DataTransConv*, *instrnum*, *algeb_connectivity*, *#id_to_constants*, and *average_degree* as shown in Listing 1.1.

Listing 1.1: `_strstr` best feature selection

```
#DataTransConv > 32.500
| instrnum > 237: other {_strstr=0,other=69}
| instrnum ? 237: _strstr {_strstr=11,other=0}
#DataTransConv ? 32.500
| algebraic_connectivity > 2.949
| | #id_to_constants > 3
| | | average_degree > 2.847: other {_strstr=0,other=591}
| | | average_degree ? 2.847
| | | |   instrnum > 171: _strstr {_strstr=1,other=0}
| | | |   instrnum ? 171: other {_strstr=0,other=48}
| | #id_to_constants ? 3: _strstr {_strstr=2,other=0}
| algebraic_connectivity ? 2.949:other{_strstr=0,other=4745}
```

4 Detection

Given a target binary function, the disassembled function is passed through two filters to obtain a set of candidate functions from the repository. The classical approach to detection would be to employ the closest Euclidean distances [16] between all top-ranked features of target function and candidate functions. However, such an approach may not be scalable enough for handling millions of functions. Therefore, we design a novel data structure, called B^{++}tree, to efficiently organize the signatures of all the functions, and to find the best matches. Our experimental results (Section 5) will confirm the scalability of our method.

4.1 B^{++}tree Data Structure

Due to the growing number of free open-source libraries and the fact that binaries and malware are becoming bigger and more complex, indexing the signatures

of millions of library functions to enable efficient detection has thus become a demanding task. One classical approach is to store $(key, value)$ pairs as well as the indices of best features; however, the time complexity of indexing/detection would be $O(n)$, where n is the number of functions in the repository. To reduce the time complexity, we design a data structure, called B^{++}tree, that basically indexes the best feature values of all library functions in the repository in separate B^{+}trees, and links those B^{+}tree to corresponding features and functions. We also augment the B^{+}tree structure by adding *backward* and *forward* sibling pointers attached to each leaf node, which points to the previous and next leaf nodes, respectively. The number of neighbors is obtained by a user-defined *distance*. Consequently, slight changes in the values that might be due to the compiler effects or the slight changes in the source code is captured by the modified structure. Therefore, the indexing/detection time complexity will be reduced to $O(log(n))$, which is asymptotically better. It is worth noting that the B^{+}tree could be replaced with similar data structures such as red-black tree [9].

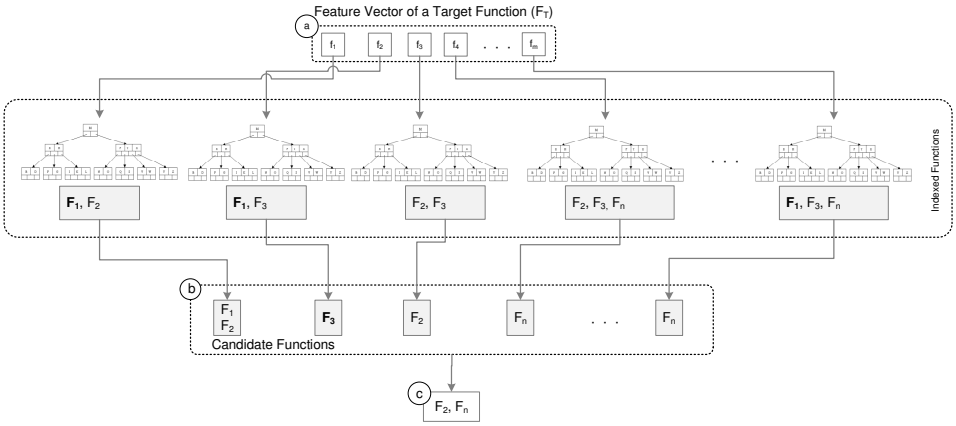

Fig. 4: Indexing and detection structure

We explain the B^{++}tree structure with a small example illustrated in Figure 4. The best features of all library functions in our repository are indexed in B^{+}trees depicted in the middle box. For instance, the best features of library function F_1 are f_1, f_2 and f_m; hence, these three feature values linked to the function F_1 are indexed in the corresponding B^{+}trees (shown in boldface). For the purpose of detection, (a) all the features of a given target function are extracted. For each feature value, a lookup is performed on the corresponding B^{+}tree, and (b) a set of candidate functions based on the closest values and the user-defined *distance* are returned (we assume that $\{F_1, F_2, F_3, F_n\}$ are returned as the set of candidate functions). For instance, one match is found for the f_2 feature with the second feature of function F_3 (shown in boldface in part b),

whereas this feature is indexed as the best feature for F_1 function as well. Finally, the candidate functions are sorted based on the distance and total number of matches: $\{F_2, F_n, F_1, F_3\}$. If we consider the first most frequent functions ($t = 1$), the final candidates would be $\{F_2, F_n\}$ functions.

The details are shown in Algorithm 1. Let f_T be the target function, and F retains the set of candidate functions and their frequency as output. First, all top-ranked features are extracted from the given target binary f_T (line 6). By performing m (total number of features) lookups in each B^+trees (line 7), a set of candidate functions will be returned (line 8). In order to choose the top t functions, the most t frequent functions are returned as the final set of matched functions (line 10).

Algorithm 1: Function Detection

Input: f_T : Target function.
Output: F : Set of candidate functions.
Initialization
1 $m \leftarrow$ total number of features;
2 $n \leftarrow$ total number of functions in the repository;
3 $F_c \leftarrow \{\}$; dictionary of candidate functions ;
4 $t \leftarrow$ number of most frequent functions to be considered;
5 $distance \leftarrow$ user-defined distance;
6 $feature[m] \leftarrow$ array of size m to hold all the extracted features;
begin
7 feature[] = featureExtraction(f_T);
8 **foreach** $feature[i] \subset F_T$ **do**
9 $F_c = F_c + B^+TreeLookup(feature[i], distance)$;
10 **end**
11 $F = t\_most\_frequent\_functions(F_c, t)$;
12 **return** F;
13 **end**

4.2 Filtration

To address the scalability issue of dealing with large datasets, a filtration process is necessary. Instead of a pairwise comparison, we prune the search space by excluding functions that are unlikely to be matched. In the literature, discovRe [17] applies the k-Nearest Neighbors algorithm (kNN) on numerical features as a filter to find similar functions. However, Genius [19] re-evaluates discovRe, and illustrates that pre-filtering significantly reduces the accuracy of discovRe. To this end, two simple filters are used in our work, which are described hereafter.

Basic Blocks Number Filter (BB#) It is unlikely that a function with four basic blocks can be matched to a function with 100 basic blocks. In addition, due

to the compilation settings and various versions of the source code, there exist some differences in the number of basic blocks. Thus, a user-defined threshold value (γ) is employed, which should not be too small or too large to prevent discarding the correct match. Therefore, given a target function f_T, the functions in the repository which have $\gamma\%$ more or less basic blocks than the f_T are considered as candidate functions for the final matching. Based on our experiences with our dataset, we consider $\gamma = \pm 35$.

Instruction Number Filter (Ins#) Similarly, given a target function f_T, the differences between the number of instructions of target function f_T and the functions in the repository are calculated; if the difference in the number of instructions is less than a user-defined threshold value λ, then the function is considered as a candidate function. According to our dataset and experiments, we consider $\lambda = 35\%$.

5 Evaluation

In this section, we present the evaluation results of proposed technique. First, we explain the environment setup details followed by the dataset description. Then, the main accuracy results of library function identification are presented. Furthermore, we study the impact of different obfuscation techniques as well as the impact of compilers on the proposed approach and discuss the results. Additionally, we examine the effect of feature selection on our accuracy results. We then evaluate the scalability of *BinShape* on a large dataset. Finally, we study the effectiveness of *BinShape* on a real malware binary.

5.1 Experiment Setup

We develop a proof-of-concept implementation in python to evaluate our technique. All of our experiments are conducted on machines running Windows 7 and Ubuntu 15.04 with Core i7 $3.4GHz$ CPU and $16GB$ RAM. The described approach is implemented using separate python scripts for modularity purposes, which altogether form our analytical system. The Matlab software has been used for the graph feature extraction. A subset of python scripts in the proposed system is used in tandem with IDA Pro disassembler. The *MongoDB* database [3] is utilized to store our features for efficiency and scalability purposes. For the sake of usability, a graphical user interface in which binaries can be uploaded and analyzed is implemented.

Any particular selection of data may not be representative of another selection. Hence, to mitigate the possibility that results may be biased by the particular choice of training and testing data, a $C4.5(J48)$ decision tree is evaluated on a $90:10$ training/test split of the dataset.

5.2 Dataset Preparation

We evaluate our approach on a set of binaries, as detailed in Table 3. In order to create the ground truth, we download the source code of all C-library functions [1], as well as different versions of various open-source applications, such as 7-zip, Notepad++, Python, etc. The source code are compiled with Microsoft Visual Studio (VS 2010, and 2012), and GNU Compiler Collection (GCC 4.1.2) compilers, where the /MT and -static options, respectively, are set to statically link C/C++ library. In addition, the $O0 - O3$ options are used to examine the effects of optimization settings. Program debug databases (PDBs) holding debugging information are also generated for the ground truth. Furthermore, we obtain binaries and corresponding PDBs from their official websites (e.g., WireShark); the compiler of these binaries are detected by a packer tool called ExeinfoPE [2].

The prepared dataset is used as the ground truth for our system, since we can verify our results by referring to source code. In order to demonstrate the effectiveness of our approach to identify library functions in malware binaries, we additionally choose `Zeus` malware version 2.0.8.9, where the source code was leaked in 2011 and is reused in our work[1].

Project	Version	No. Fun.	Size(Kb)	Project	Version	No. Fun.	Size(Kb)
7zip/7z	15.14	133	1074	nspr	4.10.2.0	881	181
7zip/7z	15.11	133	1068	nss	27.0.1.5156	5979	1745
7-Zip/7zg	15.05 beta	3041	323	openssl	0.9.8	1376	415
7-Zip/7zfm	15.05 beta	4901	476	avgntopensslx	14.0.0.4576	3687	976
bzip2	1.0.5	63	40.0	pcre3	3.9.0.0	52	48
expat	0.0.0.0	357	140	python	3.5.1	1538	28070
firefox	44.0	173095	37887	python	2.7.1	358	18200
fltk	1.3.2	7587	2833	putty/putty	0.66 beta	1506	512
glew	1.5.1.0	563	306	putty/plink	0.66 beta	1057	332
jsoncpp	0.5.0	1056	13	putty/pscp	0.66 beta	1157	344
lcms	8.0.920.14	668	182	putty/psftp	0.66 beta	1166	352
libcurl	10.2.0.232	1456	427	WireShark/Qt5Core	2.0.1	17723	3987
libgd	1.3.0.27	883	497	SQLite	2013	2498	1006
libgmp	0.0.0.0	750	669	SQLite	2010	2462	965
libjpeg	0.0.0.0	352	133	SQLite	11.0.0.379	1252	307
libpng	1.2.51	202	60	TestSSL	4	565	186
libpng	1.2.37	419	254	tinyXML	2.0.2	533	147
libssh2	0.12	429	115	Winedt	9.1	87	8617
libtheora	0.0.0.0	460	226	WinMerge	2.14.0	405	6283
libtiff	3.6.1.1501	728	432	WireShark	2.0.1	70502	39658
libxml2	27.3000.0.6	2815	1021	WireShark/libjpeg	2.0.1	383	192
Notepad++	6.8.8	7796	2015	WireShark/libpng	2.0.1	509	171
Notepad++	6.8.7	7768	2009	xampp	5.6.15	5594	111436

Table 3: An excerpt of the projects included in our the dataset

5.3 Function Identification Accuracy Results

Our ultimate goal is to discover as many relevant functions as possible with less concern about false positives. Consequently, in our experiments we use the F-measure, $F_1 = 2 \cdot \frac{P \cdot R}{P+R}$, where P is the precision, and R is the recall. Additionally,

[1] https://github.com/Visgean/Zeus

we show the ROC curve for each set of features used by *BinShape*. To evaluate our system, we split the binaries in the ground truth into ten sets, reserving one as a testing set and using the remaining nine as training sets. We repeat this process 1000 times and report the results that are summarized in Figure 5.

We obtain a slightly higher true positive rate when using graph features (including function-call graph feature) and statistical features. This small difference can be inferred due to the graph similarity between two library functions that are semantically close. Similarly, statistical features convey information related to the functionality of a function, which cause a slight higher accuracy. On the other hand, instruction-level features return lower true positive rate. However, when all the features are combined together, our system returns an average F_1 measure of 0.89.

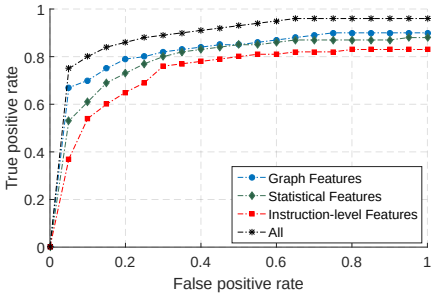

Fig. 5: ROC curve for *BinShap* features

5.4 Impact of Obfuscation

In the second scenario, we investigate the impact of obfuscation techniques on *BinShape* as well as FLIRT and *libv* approaches. Our choices of obfuscation techniques are based on the popular obfuscator LLVM [28] and DaLin [35], which include *register renaming* (RR), *instruction reordering* (IR), *dead code insertion* (DCI), *instruction substitution* (SUB), *control flow flattening* (FLA), and *bogus control flow* (BCF). Other obfuscation techniques, such as *code compression* and *encryption*, will be investigated in our future work.

For this purpose, we collect a random set of files (i.e., 25) compiled with compilers. The binaries are converted into assembly files through disassembler, and the code is then obfuscated using DaLin. We initially test the original selected files and report accuracy measurements. The obfuscation is then applied and new accuracy measurements are obtained. The effectiveness of obfuscation is shown in Table 4. As can be seen, *BinShape* can tolerate some obfuscation techniques. The accuracy remains the same when RR and IR techniques are applied, while it is reduced slightly in the case of DCI and SUB obfuscations. The reason is that most of the features which are not extracted from the instruction-level features (e.g., graph features), are not significantly affected by these techniques. In addition, normalizing the assembly instructions eliminates the effect of RR, whereas, statistical features are more affected by DCI and SUB techniques, since these features rely on the frequency of instruction. However, the accuracy results after applying FLA and BCF through LLVM obfuscator are not promising, and we exclude them from our experiments. Therefore, additional de-obfuscation techniques are required for handling these kinds of obfuscations.

5.5 Impact of Compilers

In this section, we examine the effects of compilers on a random subset of binaries as follows. (i) *The impact of compiler version.* We train our system with binaries compiled with VS 2010 ($O2$) and test the system with binaries compiled with VS 2012 ($O2$). (ii) *The impact of optimization levels.* We train our system with binaries compiled with VS 2010 at optimization level $O1$, and test it under the same compiler at optimization level $O2$. (iii) *The impact of different compilers.* We collect binaries compiled with VS 2010 ($O2$) as training dataset, and test the system with binaries compiled with GCC 4.1.2 compiler ($O2$). The obtained precision and recall for the scenarios are reported in Table 5. We observe that our system is not affected significantly by changing the compiler versions. Also, the accuracy drops slightly when the optimization level is changed. However, different compiler families affect the accuracy. Examining the effects of more possible scenarios, such as comparing binaries compiled with the same compiler but at optimization levels $O1$ and $O3$, is one of the subjects of our feature work.

Obfus. Tech.	*BinShape* $F_1 = 0.89$ F_1^*	FLIRT $F_1 = 0.81$ F_1^*	*libv* $F_1 = 0.84$ F_1^*
RR	0.89	0.81	0.84
IR	0.89	0.78	0.82
DCI	0.88	0.80	0.82
SUB	0.86	0.79	0.80
All	0.86	0.76	0.80

Table 4: F-measure before/after (F_1/F_1^*) applying obfuscation

Project	Version		Optimization		Family	
	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
`bzip2`	1.00	0.98	0.90	0.85	0.82	0.8
`OpenSSL`	0.93	0.78	0.91	0.80	0.83	0.78
`Notepad++`	0.98	0.97	0.95	0.82	0.84	0.72
`libpng`	1.00	1.00	0.91	0.74	0.81	0.72
`TestSTL`	0.98	1.00	0.90	0.84	0.81	0.75
`libjpeg`	0.93	0.90	0.88	0.76	0.81	0.69
`SQLite`	0.91	0.87	0.89	0.85	0.78	0.71
`tinyXML`	1.00	0.99	0.90	0.82	0.84	0.79

Table 5: Impact of compilers

5.6 Impact of Feature Selection

We carry out a set of experiments to measure the impact of feature selection process, including top-ranked feature selection as well as best feature selection. First, we test our system to determine the best threshold value for top-ranked features as shown in Figure 6. We start by considering five top-ranked featrures and report the F_1 measure of 0.71. We increment the number of top-ranked features by five each time. When the number of top-ranked features reaches 35 classes, the F_1 measure is increased to 0.89 and it remains almost constant afterwards. Based on our findings, we choose 35 as the threshold value for the top-ranked feature classes.

Next, we pass the top-ranked features into the decision tree in order to select the best features for each function. We aim to investigate whether considering the subset of best features would be enough to segregate the functions. In order to examine the effect of best features, we perform a breadth first search (BFS) on the corresponding trees to sort best features based on their importance in the function; since the closer the feature is to the root, the more it is segregative.

Our experiments examine the F_1 measure while varying the percentage of best features. We start by 40% of the top-ranked best features and increment them by 10% each time. Figure 7 shows the relationship between the percentage of best features and the F_1 measure. Based on our experiments, we find that 90% of the best features results in an F_1 measure of 0.89. However, for the sake of simplicity, we consider all the selected best features in our experiments.

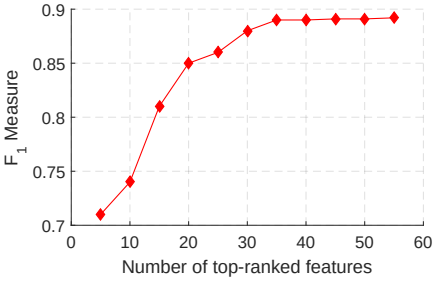

Fig. 6: Impact of top-ranked features

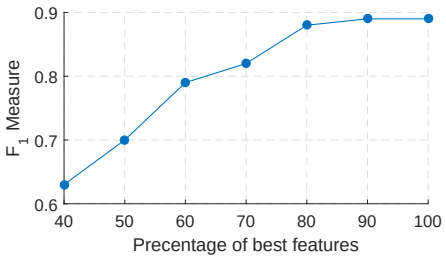

Fig. 7: Impact of best features

5.7 Impact of Filtration

We study the impact of the proposed filter (e.g., $BB\#$ and $Ins\#$) on the accuracy of *BinShape*. For this purpose, we perform four experiments by applying: (i) no filtering (ii) $BB\#$ filter, (iii) $Ins\#$ filter, and (iv) the two filters. As shown in Figure 8, the drop in the accuracy caused by the proposed filters is negligible. For instance, when we test our system with two filters, the highest drop in accuracy is about 0.017. We observe through this experiment that $Ins\#$ filter affects accuracy more than $BB\#$ filter.

5.8 Scalability Study

To evaluate the scalability of our system, we prepare a large collection of binaries consisting of different '.exe' or '.dll' files (e.g., `msvcr100.dll`) containing more than $3,020,000$ disassembled functions. We gradually index this collection of functions in a random order, and query the `7-zip` binary file of version 15.14 on our system at an indexing interval of every $500,000$ assembly functions. We collect the average indexing time for each function to be indexed, as well as the average time it takes to respond to a function detection. The indexing time includes feature extraction and storing them in the B^+trees. Figure 9 depicts the average indexing and detection time for each function.

The results suggest that our system scales well with respect to the repository size. When the number of functions in the repository increases from $500,000$ to $3,020,000$, the impact on response time of our system is negligible (0.14 seconds on average to detect a function amongst three million functions in the repository). We notice through our experiments that the ranking time and filtering time are very small and negligible. For instance, ranking $5,000$ features takes

0.0003 ms, and to filter high likely similar functions in the repository to a function having 100 basic blocks and 10,000 instructions, takes 0.009 ms. Besides, the feature extraction time varies for different types of features; e.g., the graph feature extraction takes more time than instruction-level and call-graph feature extractions.

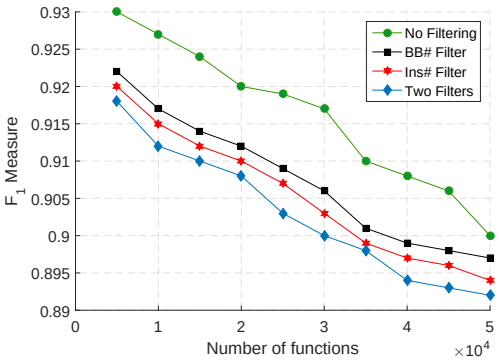

Fig. 8: Impact of filtration

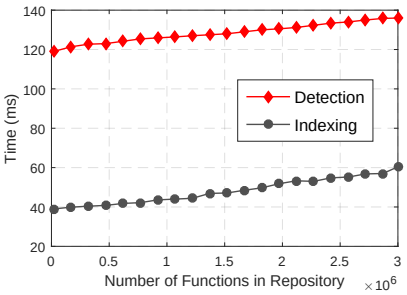

Fig. 9: Scalability study

5.9 Application to Real Malware

We are further interested in studying the practicability and applicability of our tool in identifying library functions in malware binaries. However, one challenge is the lack of ground truth to verify the results due to the nature of malware. Consequently, we consider `Zeus` version 2.0.8.9, where the leaked source code is available. First, we compile the source code with VS and GCC compilers, and keep the debug information for the purpose of verification. Second, we compile `UltraVNC`, `info-zip`, `xterm` and `BEAEngine` libraries with VS and GCC compilers and index inside our repository. We choose the aforementioned libraries based on the technical report [30] that reveals which software components are reused by `Zeus`. Finally, we test the compiled binaries of `Zeus` to find the similar functions with the functions in our repository. By manually examining the source code as well as the debug information at binary level, we are able to verify the results listed in Table 6. We observe through our experiments that the statistical features as well as graph features are the most powerful features in discovering free open-source library functions.

Library	No. of Functions	*BinShape*	
		Found	FP
`UltraVNC`	20	28	11
`info-zip`	30	27	0
`xterm`	17	18	2
`BEAEngine`	21	20	0

Table 6: Library function identification in `Zeus`

6 Related Work

Various approaches for library function identification have been proposed. The well-known FLIRT [6] technique builds the signatures from the first 32 bytes of a function with wildcards for bytes that vary when the library is loaded. However, the main limitations of FLIRT are signature collision as well as requiring a new signature even for slight changes in a function. UNSTRIP [26] identifies wrapper functions in the GNU C library based on their interaction with the system call interface. A semantic descriptor based on the name and any concrete parameter values of the invoked system call as a fingerprint is constructed. Then, a flexible pattern matching algorithm to identify the wrapper functions based on library fingerprints is used. However, UNSTRIP is limited to the wrapper functions which invoke a system call. Another approach called libv [41], introduces execution dependence graphs (EDGs) to describe the behavioral characteristics of binary code. Then, by applying a graph isomorphism and finding similar EDG subgraphs in target functions, both full and inline library functions are identified. However, various library functions may have the same EDGs. In addition, compiler optimization might affect the instructions of inline functions, which may not be identified. The well-known BinDiff [12] technique and BinSlayer [8] (which is inspired by BinDiff), perform a graph isomorphism on functions pairs in two differing but similar binaries. However, these approaches are not designed to label library functions and rely on graph matching, which is expensive to be applied to large scale datasets.

Moreover, a few binary code search engines have been proposed. For instance, Rendezvous [31] extracts multiple features including mnemonics, n-grams, control flow subgraphs and data constants to form the tokens. The tokens are processed to form query terms in order to construct the search query. However, this approach is sensitive to structural changes. TRACY [11] decomposes the CFG into small tracelets and uses longest common subsequence (LCS) algorithm to align two tracelets. However, since much structure information may be lost by breaking down the CFGs into tracelets, the authors believe TRACY is only suitable for the functions with more than 100 basic blocks [11].

An obfuscation-resilient method called CoP [37] combines symbolic execution with longest common subsequence to compare two functions in program binaries. The semantics of a basic block is modeled by a set of symbolic formulas representing the input-output relations of the block. However, since symbolic execution typically has a relatively high computational overhead, the approach may not be practical for large datasets. Inspired by CoP, BinSequence [25] compares two functions using longest common subsequence and neighborhood exploration. However, the accuracy of BinSequence drops due to the effects of code transformation, e.g., instruction reordering. In addition, if the filtration process does not filter significant number of functions, the time complexity could be degraded significantly.

PROPOSALS	Feature				Ana.		Method	Arch.			Compiler				Obfusc.			
	Syntactic	Semantic	Structural	Statistical	Static	Dynamic		x86-64	ARM	MIPS	VS	GCC	ICC	Clang	RR	IR	DCI	SUB
FLIRT [6]	•				•		PM	•	•	•	•	•			•			
UNSTRIP [26]		•			•		PM, SE	•				•			•	•	•	•
libv [41]		•	•		•		GI, DFA	•			•				•	•	•	•
BinDiff [12]		•	•		•		GI, SPP	•	•	•	•	•			•	•		
BinSlayer [8]		•	•		•		GI, GED	•	•			•			•	•		
CoP [37]		•	•		•		LCS, SE	•				•	•		•	•	•	
BinSequence [25]	•		•		•		LCS	•			•				•			
Rendezvous [31]	•		•		•		GI	•				•				•		
TRACY [11]		•	•		•		LCS, DFA	•				•			•	•	•	
BLEX [15]		•				•	JD	•				•	•	•	•	•	•	•
discovRE [17]			•	•	•		MCS, JD	•	•	•					•	•		
Genius [19]			•	•	•		VLAD, LSH, JD	•	•	•		•		•	•	•		
BinShape	•	•	•	•	•		B^{+}tree	•			•	•			•	•	•	•

Table 7: Comparing different existing solutions with *BinShape*. The symbol (•) indicates that proposal offers the corresponding feature, otherwise it is empty. We use following abbreviations for different methods: (PM) Pattern Matching, (GI) Graph Isomorphism, (GED) Graph Edit Distance, (LSH) Locality Sensitive Hashing, (LCS) Longest Common Subsequence, (MCS) Maximum Common Subgraph Isomorphism, (SPP) Small Primes Product , Symbolic Execution (SE), Data Flow Analysis (DFA), Jaccard Distance (JD), VLAD (Vector of Locally Aggregated Descriptors) .

Some recent cross-architecture approaches for searching known bugs in firmware images have been proposed. For instance, discovRE [17] extracts statistical and structural features, and then applies maximum common subgraph (MCS) isomorphism on the CFGs to find similar functions. This approach utilizes pre-filtering in order to perform subgraph isomorphism efficiently. However, Genius [19] evaluates the effectiveness of the pre-filtration of discovRE [17] and demonstrates its unreliability, which can cause significant reduction in accuracy. Inspired by discovRE, Genius [19] extracts statistical and structural features, generates codebooks from annotated CFGs, converts the codebooks into high-level numeric feature vectors (feature encoding), and finally compares the encoded features using locality sensitive hashing (LSH) in order to tackle scalability issues. However, the authors mentioned that codebook generation may be expensive [19]. In addition, changes in the CFG structure affect the accuracy of Genius [19]. Moreover, signature collision might be an issue, since similar and especially small functions high likely will have similar hash values.

All of the aforementioned approaches employ static analysis, however, there exist other techniques which perform dynamic analysis. For instance, a binary search engine called Blanket Execution (BLEX) [15], executes functions for several calling contexts and collects the side effects of functions; two functions with similar side effects are deemed to be similar. However, dynamic analysis approaches are often computationally expensive, and rely on architecture-specific

tools to run executables. As a result, they are inherently difficult to support other architectures [17].

We compare *BinShape* with existing proposals in Table 7 in terms of features, types of analysis (denoted as Ana.), methodology, supported architecture (denoted as Arch.), compilers, and finally whether the approach is robust against some obfuscation techniques, such as register renaming (RR), instruction reordering (IR), dead code insertion (DCI), and equivalent code substitution (SUB). As can be seen, among the existing works, libv [41], CoP [37], BinSequence [25], discovRE [17], and Genius [19] involve computationally expensive methods (e.g., GI); BinDiff [12] and BinSlayer [8] are designed for individual projects, and do not support searching in a large number of projects; BLEX [15] is a dynamic analysis approach, which cannot be extended easily to support other architectures; UNSTRIP [26] is limited to wrapper functions; FLIRT [6] matches signatures generated from syntactic features, and is sensitive to signature collision and slight changes in the byte sequences. In contrast, *BinShape* does not involve expensive computation methods, and it supports large scale of dataset. It extracts rich set of heterogeneous features, which could overcome some code changes and it is not limited to wrapper functions. Finally, *BinShape* can be extended easily to support other compilers or architectures.

A direct comparison between Genius [19] and *BinShape* is not possible, since the authors have not made their Genius prototype available. However, a high-level comparison has been done as follows. Genius is designed to identify known bugs in firmware images, while the goal of *BinShape* is to identify library functions in program binaries and malware samples. Genius extracts few statistical and structural features, while a richer set of features (in terms of types and numbers) are extracted by *BinShape*. Genius detects a function out of three million functions in about 0.007 seconds, whereas 0.14 seconds takes for *BinShape* to detect a function among three million functions indexed in the repository. However, the time given to create the ground truth (which includes codebook generation) in Genius cannot be compared with *BinShape*, since our criteria is based on the number of functions, which is not provided by Genius. Based on the provided ROC curve in Genius [19], we have obtained the average true positive rates of 0.96, while that of *BinShape* is 0.91.

7 Conclusion

Limitations. Our work has the following limitations.

- *Function Inlining*: The compiler may inline a small function into its caller code for optimization purposes. Therefore, fingerprinting a function with partial code in another function would be required. However, *BinShape* does not currently support function inlining.
- *Advanced Obfuscation Techniques*: Our system is able to tackle some code transformation such as instruction reordering, and dead code insertion. However, some other obfuscation techniques, such as control flow flattening, affect

the accuracy of *BinShape*. In addition, our system is not able to handle the packed, encrypted, and obfuscated binaries.
- *Supported Compilers*: We have tested *BinShape* with VS and GCC compilers, and reported the accuracy results. However, we have not examined binaries compiled with ICC and Clang compilers.
- *Hardware Architecture*: Our proposed system deals with only x86 architecture, and we have not investigated the impact of hardware architectures such as MIPS in this study.

These limitations are the subjects of our future work.

Concluding Remarks. In this paper, we have conducted the first investigation into the possibility of representing a function based on its shape. We have proposed a robust signature for each library function based on diverse collection of heterogeneous features, covering CFGs, instruction-level characteristics, statistical features, and function-call graphs. In addition, we have designed a novel data structure, which includes B^+tree, in order to efficiently support accurate and scalable detection. Experimental results have been demonstrated the practicability of our approach.

8 Acknowledgment

We would like to thank our shepherd, Dr. Cristiano Giuffrida, and the anonymous reviewers for providing us valuable comments. This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) of Concordia University, Defence Research and Development Canada (DRDC) Valcartier and Google under a DND/NSERC Research Partnership Program.

References

1. C Language Library, 2014. http://www.cplusplus.com/reference/clibrary/.
2. Exeinfo PE, 2014. http://exeinfo.atwebpages.com.
3. MongoDB, 2015. https://www.mongodb.com/.
4. NIST/SEMATECH e-Handbook of Statistical Methods, 2015. http://www.itl.nist.gov/div898/handbook/.
5. WEKA, 2015. https://weka.wikispaces.com/.
6. HexRays: IDA F.L.I.R.T. Technology, 2016. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.
7. HexRays: IDA Pro, 2016. https://www.hex-rays.com/products/ida/index.shtml.
8. M. Bourquin, A. King, and E. Robbins. BinSlayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.
9. T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.
10. Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 266–280. ACM, 2016.
11. Y. David and E. Yahav. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM, 2014.

12. T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.
13. C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.
14. M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.
15. M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *Usenix Security*, pages 303–317, 2014.
16. K. L. Elmore and M. B. Richman. Euclidean distance as a similarity metric for principal component analysis. *Monthly Weather Review*, 129(3):540–549, 2001.
17. S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovRe: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23th Symposium on Network and Distributed System Security (NDSS)*, 2016.
18. M. R. Farhadi, B. C. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 15:46–60, 2015.
19. Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 480–491. ACM, 2016.
20. E. Frank, Y. Wang, S. Inglis, G. Holmes, and I. H. Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.
21. H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.
22. C. Griffin. Graph theory: Penn state math 485 lecture notes. http://www.personal.psu.edu/cxg286/Math485.pdf, 2011-2012.
23. S. Hido and H. Kashima. A linear-time graph kernel. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*, pages 179–188. IEEE, 2009.
24. X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.
25. H. Huang, A. M. Youssef, and M. Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 155–166. ACM, 2017.
26. E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.
27. L.-P. Jing, H.-K. Huang, and H.-B. Shi. Improved feature selection approach tfidf in text mining. In *Machine Learning and Cybernetics, 2002. Proceedings. 2002 International Conference on*, volume 2, pages 944–946. IEEE, 2002.
28. P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, 2015.
29. M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 46–53. ACM, 2007.
30. W. M. Khoo. Decompilation as search. Technical report, University of Cambridge, Computer Laboratory, 2013.

31. W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.
32. C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 29–44. IEEE, 2010.
33. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.
34. M. Kührer, C. Rossow, and T. Holz. Paint it black: Evaluating the effectiveness of malware blacklists. In *International Workshop on Recent Advances in Intrusion Detection*, pages 1–21. Springer, 2014.
35. D. Lin and M. Stamp. Hunting for undetectable metamorphic viruses. *Journal in computer virology*, 7(3):201–214, 2011.
36. L. Livi and A. Rizzi. The graph matching problem. *Pattern Analysis and Applications*, 16(3):253–283, 2013.
37. L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400. ACM, 2014.
38. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 431–441. IEEE, 2007.
39. H. Peng, F. Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on pattern analysis and machine intelligence*, 27(8):1226–1238, 2005.
40. J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.
41. J. Qiu, X. Su, and P. Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, 2016.
42. B. B. Rad, M. Masrom, and S. Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. In *e-Learning and e-Technologies in Education (ICEEE), 2012 International Conference on*, pages 209–213. IEEE, 2012.
43. M. Ramaswami and R. Bhaskaran. A study on feature selection techniques in educational data mining. *arXiv preprint arXiv:0912.3924*, 2009.
44. D. Roobaert, G. Karakoulas, and N. V. Chawla. Information gain, correlation and support vector machines. In *Feature Extraction*, pages 463–470. Springer, 2006.
45. N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *Computer Security–ESORICS 2011*, pages 172–189. Springer, 2011.
46. A. H. Toderici and M. Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013.
47. V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 934–953. IEEE, 2016.
48. A. Yokoyama, K. Ishii, R. Tanabe, Y. Papa, K. Yoshioka, T. Matsumoto, T. Kasama, D. Inoue, M. Brengel, M. Backes, et al. Sandprint: Fingerprinting mal-

ware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 165–187. Springer, 2016.

49. J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 487–498. ACM, 2013.
50. E. R. Ziegel. Probability and statistics for engineering and the sciences. *Technometrics*, 2012.