FOSSIL: A Resilient and Efficient System for Identifying FOSS Functions in Malware Binaries

Saed Alrabaee, Paria Shirani, Lingyu Wang, Mourad Debbabi
Concordia University
Montreal, Canada

Identifying free open-source software (FOSS) packages on binaries when the source code is unavailable is important for many security applications, such as malware detection, software infringement, and digital forensics. This capability enhances both the accuracy and the efficiency of reverse engineering tasks by avoiding false correlations between irrelevant code bases. Although the FOSS package identification problem belongs to the field of software engineering, conventional approaches rely strongly on practical methods in data mining and database searching. However, various challenges in the use of these methods prevent existing function identification approaches from being effective in the absence of source code. To make matters worse, the introduction of obfuscation techniques, the use of different compilers and compilation settings, and software refactoring techniques has made the automated detection of FOSS packages increasingly difficult. With very few exceptions, the existing systems are not resilient to such techniques, and the exceptions are not sufficiently efficient.

To address this issue, we propose *FOSSIL*, a novel resilient and efficient system that incorporates three components. The first component extracts the syntactical features of functions by considering opcode frequencies and applying a hidden Markov model statistical test. The second component applies a neighborhood hash graph kernel to random walks derived from control flow graphs, with the goal of extracting the semantics of the functions. The third component applies z-score to the normalized instructions to extract the behavior of instructions in a function. The components are integrated using a Bayesian network model which synthesizes the results to determine the FOSS function. The novel approach of combining these components using the Bayesian network has produced stronger resilience to code obfuscation.

We evaluate our system on three datasets including real-world projects whose use of FOSS packages is known, malware binaries for which there are security and reverse engineering reports purporting to describe their use of FOSS, and a large repository of malware binaries. We demonstrate that our system is able to identify FOSS packages in real-world projects with a mean precision of 0.95 and with a mean recall of 0.85. Furthermore, *FOSSIL* is able to discover FOSS packages in malware binaries that match those listed in security and reverse engineering reports. Our results show that modern malware binaries contain $0.10 - 0.45$ of FOSS packages.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security, Information Systems

Additional Key Words and Phrases: Binary code analysis, malicious code analysis, function fingerprinting, free software packages

ACM Reference Format:

1. INTRODUCTION

When analyzing malware binaries, reverse engineers often pay special attention to reused FOSS packages for several reasons. First, recent reports from anti-malware companies indicate that finding the similarity between malware samples attributable to reused FOSS packages can aid in developing profiles for malware families [Kas 2016]. For instance, `Flame` [Bencsáth et al. 2012a] and other malware in its family [Bencsáth et al. 2012b] all contain code packages that are publicly available, including SQLite and LUA [Kas 2016]. Second, a significant proportion of most modern malware consists of FOSS packages; therefore, identifying reused packages is a critical preliminary step in the process of extracting information about the functionality of a malware binary. Third, in more challenging cases where obfuscation techniques may have been applied and the reused FOSS packages may differ from their original source files, it is still desirable to determine which part of the malware binary is borrowed from which FOSS packages. Fourth, in addition to identifying FOSS packages, clustering FOSS functions based on their common origin may help reverse engineers to identify new malware from a known family or to decompose a malware binary based on the origin of its functions.

To the best of our knowledge, there has been little effort devoted specifically to identifying FOSS packages in malware binaries. However, existing techniques may be applied to binaries, including binary search engines [David and Yahav 2014; Khoo et al. 2013; Chandramohan et al. 2016; Nataraj et al. 2013; Ding et al.]; determining reused algorithms [Alrabaee et al. 2015]; discovering shared components [Ruttenberg et al. 2014], clone detection [Farhadi et al. 2014; Sæbjørnsen et al. 2009], labeling standard library function [Qiu et al. 2016; Jacobson et al. 2011; Guilfanov 1997], and labeling standard compiler libraries [Jacobson et al. 2011; Rahimian et al. 2015]. However, such techniques share several limitations, including inaccuracy caused by different compilation settings, high computational overhead, and being vulnerable to obfuscation and refactoring techniques. The consequence is variability in precision/recall [Shapiro and Horwitz 1997]. For example, some matches may be incorrectly labeled as identical when they are identical only in structure but not in semantics [Stojanović et al. 2015]. Furthermore, if the original source code of the packages is modified before reusing, the existing systems will not be able to capture the resulting variations in the features upon which they rely.

Challenges. In automating the process of identifying FOSS functions in malware binaries, several challenges are typically encountered. The first is *usability*. Immediate insights obtained about a binary file from a system to highlight FOSS packages will give reverse engineers a direction to start their investigations. The existing approaches for the purpose of binary search engine, clone detection, or function identification return the top-ranked candidate functions, while these results are helpful if the repository contains a function that exhibits a high degree of similarity to the target function. Moreover, because of the effect of different compilers, compiler optimization, and obfuscation techniques, a given unknown function is less likely to be very similar to the right function in the repository, and there is little advantage in returning a list of matches with low degrees of similarity. A resilient system should be able to identify the matched pairs with a controller process that can synthesize the available knowledge. The second challenge is *efficiency*. An efficient system can help reverse engineers to find matches on the fly. 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 is challenging. The third challenge is *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. The fourth challenge is *scalability*. Reverse engineers deal with large numbers of binaries on a daily basis, so it is necessary to design a system that could scale up millions of binary functions. Accordingly, it is important to consider the factors that may degrade the performance of FOSS package identification as the repository size increases. The fifth challenge is *stability*. One of the most important concerns in the design of a system is to provide a component to update the repository, when a new version of a FOSS package is released. The update process should be supported by a system that does not need to re-index the whole package.

***FOSSIL* Overview.** To address the limitations and challenges discussed above, we propose a new system that integrates a range of syntactical, semantic, and behavioral features using Bayesian network model. Specifically, we capture the syntax of a function by considering opcode frequencies, the semantics of a function by extracting the node-node interaction from a control flow graph, and the function behavior by computing the distribution of most important opcodes. Our detection system is controlled through a Bayesian network (BN) model which synthesizes the outcomes of these three components to determine the FOSS function. In the first component, we combine a hidden Markov model (HMM) [Toderici and Stamp 2013] with a statistical test based on the chi-squared [Filiol and Josse 2007] distance to improve the detection method. In the second component, we employ a Hash Subgraph Pairwise (HSP) kernel-based [Zhang et al. 2011] approach in which the key is to apply hierarchical hash labels to expose the structural information about subgraphs in linear time. In the third component, we calculate the z-score for the most important opcode density histograms. We measure the importance of opcodes by applying the mutual information, which ranks them according to their relevance to the function.

Overview of Results. In tests with 160 real projects that reuse FOSS packages, the integration of features at different abstraction layers using the BN model made it possible to identify FOSS functions with greater accuracy and efficiency than state-of-the-art systems. Our system is able to identify FOSS packages in real-world projects with a high precision and recall rate. Moreover, it is able to discover FOSS packages in malware binaries that are consistent with those listed in security and reverse engineering reports. Our results reveal that modern malware binaries reuse FOSS packages, and shows that members of a malware family often reuse the same FOSS packages. This insight can help reverse engineers understand the behavior of malware without the need to run it, and can also permit the analyst to prioritize malware samples (i.e., malware triage) according to the types of FOSS packages that have been reused.

Contributions. Our contributions are summarized as follows:

— *FOSSIL* is the first system developed to identify reused FOSS packages in malware binaries that supports multiple feature (syntactic, semantic, and structural features). Its novelty also lies in its ability to integrate the ranked opcodes, subgraph search, and function behavior. This helps reverse engineers to recognize the types of applications that a malware binary incorporates in order to characterize the malware.
— We propose an adaptive hidden Markov and Bayesian model capable of approximating the similarity between functions. This adaptive model boosts the matching search quality and yields stable results across different datasets and metrics.
— The experiments demonstrate that the proposed system can accurately identify FOSS functions with a mean precision of 0.95 and with a mean recall of 0.85. On average, the time overhead (extracting, indexing, and searching features) is 15 sec for small applications with 650 functions, and 58.6 sec for large applications with $250,000$ functions.
— The experiments also show that as well as discovering various types of FOSS packages used by malware, the proposed system provides useful insights that can give reverse engineers and security analysts a statistical knowledge of the behavior of a malware binary. Furthermore, it was found that modern malware developed after 2008 contains $0.10 - 0.45$ FOSS packages.

Road map. The remainder of this paper is organized as follows. Section 2 provides a detailed description of the threat model and system overview. Section 3 introduces *FOSSIL* in details. Section 4 presents our evaluation of the system. While the result of applying our system to malware binaries is presented in Section 5. Further discussion is highlighted in Section 6. Section 7 introduces the related work. Section 8 highlights the limitations and concluding remarks is introduced in Section 9.

2. OVERVIEW

First, we briefly describe the threat model and highlight the in-scope and out-of-scope threats of this work. We then provide an overview of our system. Finally, we present our criteria in selecting FOSS packages for evaluation.

2.1. Threat Model

Our system is designed to assist, instead of replace, reverse engineers in various use cases, such as digital forensic analysis (e.g., clustering a group of functions based on similar fingerprints) or software vulnerability disclosure (e.g., linking the code fragment of a binary to a known vulnerable/buggy function). In what follows, we further clarify the threat model and scope of this paper.

In-Scope Threats. In designing the features and methodology of our system, we have taken into consideration certain potential threats. First, adversaries may intentionally apply light obfuscation techniques 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 make such changes to evade detection. Finally, adversaries may reuse FOSS packages through modifying and adapting them to intentionally avoid detection by our system. We show how our system resists and survives these threats in Section 4.7. Furthermore, we can certainly envision many countermeasures taken by future malware writers to evade detection by our system, and hardening our system against such countermeasures will be an ongoing process.

Out-of-Scope Threats. As previously mentioned, our system is not intended to completely replace reverse engineers. Thus, the focus of our system is not on general reverse engineering tasks, such as unpacking and de-obfuscating binaries (although we later discuss how our system handles some obfuscation methods intended to evade detection by our system), but rather on discovering user functions. Our system assumes the binary is already de-obfuscated. In addition, cases where the code is encrypted in order to reduce code size or to prevent reverse engineering are also out of the scope of our system.

2.2. System Overview

The main analytical process is divided into four stages, as shown in Figure 1.

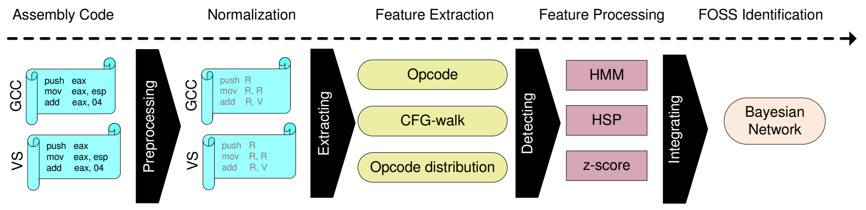


Fig. 1: An overview of the proposed system

A preprocessing stage prepares normalized disassembled instructions, followed by feature extraction once the FOSS packages are collected. Then, different detection methods are applied to the extracted features, and the repository is explored for the purpose of identification. Further, the results of these detection methods are integrated using a Bayesian network model, making it possible to identify FOSS functions and label them in the binary. In the first step, the assembly instructions are normalized (Section 3.1). The second step extracts opcodes, CFG-walks, and opcode frequency distribution features (Section 3.2). A Bayesian network controls the application of different detection methods, including HMM, HSP, and z-score, to the extracted features (Section 3.4). More specifically, the input of first method (HMM) is function opcodes, which are normalized according to the function length. A hidden Markov model (HMM) is applied to these opcode frequencies, as it can efficiently detect the behavior of a function (Section 3.4.1). The second method, HSP, accepts control flow graph walks, which are labelled by applying the kernel function for each node together with its neighbors efficiently, as described in Section 3.4.2. Depending on the output of this component, either the function is identified, or the third component of the model is checked. To achieve this, the opcode frequencies are used as input and are converted into a probability function whose characteristics are analysed with the use of z-score, as described in Section 3.4.3. These statistical features usually capture the relationship between instructions and the behavior of the function. Finally, as described in Section 4, we evaluate our approach in terms of efficiency against a set of FOSS packages compiled with different compilers and compilation settings as well as in terms of robustness against light obfuscation techniques.

2.3. FOSS Packages

Collecting FOSS packages is a crucial step in evaluating our system. To build a repository of FOSS functions, the packages are chosen using statistics that show the prevalence of FOSS libraries [Lin 2016], studies of malware behavior [Comparetti et al. 2010; Gañán et al. 2015; Yavvari et al. 2012; Ye et al. 2010], and technical reports [Kas 2016; Bencsáth et al. 2012a; Moran and Bennett 2013]. We either collect the executable files or compile these packages according to their dependencies. We tailor our system to C-family compilers because of their popularity and widespread use, especially in the development of malicious programs [Lindorfer et al. 2012]. The FOSS packages were created to perform various functionalities as partially listed in Table I.

Table I: Example of FOSS packages

Functionality	
Compression (e.g., info-zip)	MSDN libraries (e.g., NSPR)
Database management (e.g., SQLite)	Network operations (e.g., webhp)
Encryption (e.g., TrueCrypt)	Random number generation (e.g., Mersenne Twister)
File manipulation (e.g., libjsoncpp)	Secure connection (e.g., libssh2)
Hashing (e.g., Hashdeep)	Secure protocol (e.g., openssl)
Image compression (e.g., openjpeg)	Terminal emulation (e.g., xterm)
Multimedia protocols (e.g., Libavutil)	XML parser (e.g., TinyXML)

3. DESIGN AND IMPLEMENTATION OF OUR SYSTEM

In this section, we introduce our system design and describe the features in detail. We also provide an overview of the implementation environment.

3.1. Normalization

Each assembly instruction in x86 architecture 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, due to the effect of compilers and compilation settings, for instance. Hence, it is essential that the assembly code be normalized prior to comparison. Consequently, constant values and memory references are generalized by the normalizer to V and M, respectively, where simply the exact values are ignored. in Addition, registers can be generalized according to the various levels of normalization [Farhadi et al. 2015]. The top-most level generalizes all registers, regardless of type, to REG. The next level differentiates General Registers (e.g., `eax` and `ebx`), Segment Registers (e.g., `cs` and `ds`), and Index and Pointer Registers (e.g., `esi` and `edi`). The third level breaks down the General Registers by size into three groups: 32-, 16-, and 8-bit registers.

3.2. Features

In what follows, we introduce opcodes, CFG-walks, and opcode frequency distribution features used in our system.

Opcodes. Opcodes are defined as operational codes, which can be used to efficiently detect obfuscated or metamorphic malware [Bilar 2007]. However, Bilar et al. show that prevalent opcodes (e.g., `mov`, `push`, and `call`) do not make good indicators of malware samples [Bilar 2007], and based on such opcode frequencies, the resultant degree of similarity between two files could potentially be marred [de la Puerta et al. 2015]. Therefore, we propose a way to avoid this phenomenon and to give each opcode its actual relevance by applying feature ranking based on mutual information [Peng et al. 2005] in order to consider only the top-ranked opcodes.

Control Flow Graph Walks. Control Flow Graphs (CFGs) consist of a set of basic blocks, each of which represents a sequence of instructions without an intervening control transfer instruction. In the literature, CFGs have been used to detect variants of malware [Cesare et al. 2014]. Exact matching of the CFG itself does not offer much help towards our goal, since the CFG might change, for instance, due to the effect of compilers and optimization settings. Consequently, we decompose a CFG into a set of walks, taking into consideration the interactions within these walks. By doing so, we will be able to convert CFGs into a set of semantic relations (walk interactions) such that when a malware uses part of a FOSS function to implement a specific functionality, it would be captured based on these semantic relations. In the literature, the random walk kernel [Gärtner et al. 2003] and the shortest path kernel [Vishwanathan et al. 2010] are amongst the most prominent graph kernels that have been used. A graph is decomposed into sequences of nodes generated by walks; it counts the number of identical walks that can be found in two graphs. We propose an instance of the substructure fingerprint kernel suitability for the analysis of CFG walk relations.

Example: Suppose a CFG consisting of ten basic blocks ($BB_0, \cdots, BB_9$) as shown in Figure 2a. To illustrate the random walk selection, we consider two nodes BB_0 and BB_7, where the path between them (BB_0,BB_4,BB_6,BB_7) with a distance of 3 is highlighted in Figure 2b. To reduce time complexity, we consider a radius for our random walk, which is the shortest path with neighboring nodes. In our experiments, we consider radius = $\{0, 1, 2\}$, as illustrated in Figures 2b-d, respectively. By choosing the radius equal to 0, the information is only about node BB_0 and node BB_7 as shown in Figure 2b. When $r = 1$ the (BB_0, BB_4), and (BB_6, BB_7) pairs represent the structural information depicted in Figure 2c. The walks when radius is equal to 2 are (BB_0, BB_4, BB_6), and (BB_4, BB_6, BB_7). Through our experiments, we find that a radius of 2 is the best choice in terms of efficiency.

Opcode Frequency Distribution. We select the opcode frequency distribution feature based on the following observations obtained from our experiments. We first con-

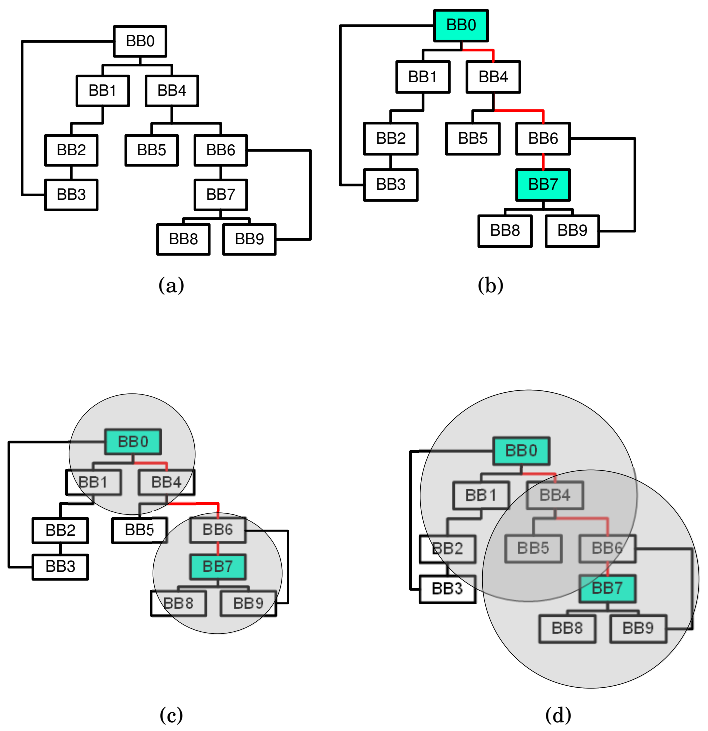


Fig. 2: An example of random walks between two nodes BB_0 and BB_7 in (a) CFG of a function, by considering three radius (r) values: (b) $r = 0$, (c) $r = 1$, and (d) $r = 2$

sider the simple hypothesis that FOSS functions performing the same task usually exhibit similar distributions of opcodes [Bilar 2007]. Second, considering the fact that the area under a frequency distribution curve is always 1, we calculate the percentage of top-ranked opcode frequencies under the distribution. Third, the distribution of various opcodes conforms to a consistent distribution shape [Das et al. 2016] when it is related to a specific FOSS function, even if the function is modified; since the semantics will be preserved [Jang et al. 2011] and may be discovered by the distribution.

3.3. Feature selection

We are interested in extracting features that represent the functionality and semantics of binary functions. We extract different representations of code properties, but only a subset of these representations may serve as indicators of the semantics of a function. Hence, we aim to select features that best preserve the semantics of a function. As such, instead of relying only on syntax-based features obtained from feature templates [Farhadi et al. 2014], we propose capturing different function features at various abstraction levels of the binary code. Furthermore, we consider how to efficiently extract features from binaries and how to efficiently store them in a repository.

Opcode ranking. The first category of our features are opcodes. By applying mutual information-based ranking [Cang and Partridge 2004] to the opcodes and corresponding functions of each FOSS family, we reduce the feature set size to effectively represent the properties of coding functionality. The opcodes with highest ranking values will be used to calculate opcode frequency distributions and to color CFG-walks.

We employ mutual information to determine the degree of statistical dependence of two variables X and Y as follows:

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

where x is the opcode frequency, y is the class of FOSS function (e.g. `sqlite3MemMalloc`), $p(x)$ and $p(y)$ are the marginal probability distribution of each random variable, and $p(x, y)$ is the joint probability of X and Y. The joint and marginal distributions are computed over the number of function variants N (For each function we have different versions such as when it is compiled with VS or GCC). These distributions are computed between class (function label) and feature as follows:

$$P(x) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[x_i=x]}, P(y) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[y_i=y]}, P(x,y) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[x_i=x \bigwedge y_i=y]}$$

Graph coloring. Relying on structural information to identify functions which are semantically similar is not sufficient given the fact that two distinct functions may still have identical CFGs [Alrabaee et al. 2015]. This shortcoming is addressed by the idea of graph coloring, where the content of each basic block is also taken into consideration. We use the graph coloring technique proposed in [Kruegel et al. 2005] to color the nodes based on the group of instructions in a basic block. This technique categorizes the instructions according to their semantics; for instance, `push` and `pop` opcodes are classified in one class (e.g., Stack operation). As a result, there are fewer possibilities for an attacker to find semantically equivalent instructions from different classes. Furthermore, the possible variations in coloring that can be generated with instructions from different classes are much fewer than the possible variations on the instruction level [Kruegel et al. 2005]. We apply the coloring technique on the normalized instructions (including the opcodes and operands) by considering only the top-ranked opcodes. Finally, we assign a weight to each edge by aggregating the colors of the source and destination nodes. For instance, if there is an edge between node A to node B, and they are colored in 2 and 8 respectively, the weight of this edge would be 10.

Opcode importance. We follow the model used in [Bilar 2007] to measure the importance of opcodes. The top-ranked opcodes are further processed through converting the frequencies into a histogram and measuring the area of intersection based on the probability distribution. This step illustrates the importance of each ranked opcode in terms of function behavior. The most important opcodes will be used by the third component. We show an example of opcode distribution values for the most important opcode that found in the introduced sorting algorithms (e.g., we consider two variants for each sorting algorithm) in Table II.

3.4. Detection method

In this section, we introduce the components of our detection system: the hidden Markov model, the neighborhood hash graph kernel, and the z-score, where the Bayesian network integrates these components.

3.4.1. Hidden Markov Model.

After the mutual information between the FOSS function and the individual opcodes is computed, an opcode relevance file based on the top-ranked features for each function is created. These top-ranked opcodes are used for the hidden Markov model (HMM) with chi-squared testing. Thus, the functions are scored (according to the opcode sequences) and classified based on whether they belong to the FOSS or not. Following

Table II: Opcode distribution values of sorting algorithms (B:Bubble sort Q:Quick sort H:Heap sort M:Merge sort)

	The z-score for the top opcode importance					
	`Add`	`Push`	`Lea`	`Move`	`And`	`Or`
`B1`	1.051211	0.412661	-0.830144	1.510222	2.100523	1.995820
`B2`	1.071291	0.406912	-0.830144	1.493213	2.215012	1.969582
`Q1`	1.059154	0.569042	-0.908245	1.618144	2.121009	1.865189
`Q2`	1.056159	0.520013	-0.900166	1.618306	2.116910	1.860028
`H1`	1.097158	0.450061	-0.731512	1.590128	2.000211	1.890244
`H2`	1.097054	0.410901	-0.699958	1.589422	2.009144	1.894061
`M1`	1.029032	0.542113	-0.700193	1.5	1.961411	1.900019
`M2`	1.018054	0.520393	-0.700082	1.5	1.959009	1.901200

this, we apply chi-squared distance with a HMM, as a way to create a confidence interval for this component. HMM was picked to be the initial component since it is computationally efficient [Toderici and Stamp 2013].

In the HMM model, the states represent the sequence of instructions, however, they are not fully observed; yet, the hidden states can be estimated by observing the sequences of data [Stamp 2004]. To discover the hidden states (e.g., instructions related to inline functions), we apply data flow analysis such as read and write dependencies as what follows. There exists a data flow dependency between two instructions i_1 and i_2 according to the following rules: (i) i_1 reads from a register or a memory address, and i_2 writes to the same register or memory address. (ii) i_1 writes to a register or memory address and, i_2 writes to the same register or memory. (iii) i_1 writes to a register or memory address, and i_2 reads from the same register or memory. Consequently, if an instruction (or set of commands) shows no evidence of a data flow dependency, it is tagged as a hidden state. It should be noted that "instruction side effect" (which flag is manipulated) are treated as observations. Therefore, such observations will be annotated to the states.

In what follows, we describe the chi-squared distance, which is combined with the HMM. The main objective is to determine the preeminent characteristics of the probability distribution of statistical opcode variable Z. The best way to find out which hypothesis is the best match for an observed sequence of samples $(Z_1, Z_2, \ldots, Z_n)$ is to use statistical testing. In fact, the Pearson's χ^2 statistic [Filiol and Josse 2007] is widely employed to confirm whether or not the discrepancies between the observed and expected data are significant. We denote this test as T^2, which is given by [Stamp 2004; Toderici and Stamp 2013]:

$$T^2 = \sum_{(i=1)}^{z} \frac{(\hat{m}_i - m_i)^2}{m_i} \leq \chi^2(\alpha, v-1) \tag{1}$$

where $\hat{m}_i$ and m_i are the normalized frequencies of opcodes in the testing phase and training phase, respectively; $(\alpha, v-1)$ represent type I error rate, and the degrees of freedom, respectively. Finally, based on the comparison results of T^2 and $\chi^2(\alpha, v-1)$, the decision threshold is acquired. For more details, we refer the reader to [Stamp 2004; Toderici and Stamp 2013].

3.4.2. Neighborhood Hash Graph Kernel.

As Gartner et al. [Gärtner et al. 2003] show, the distance between CFG-walks (node-node interaction relations) has a crucial impact on obtaining the semantics of a graph. To be more precise and to take into consideration the subgraph pairs (not only pairs), we apply a hash subgraph pairwise (HSP) [Zhang et al. 2011] kernel based method to represent the structural information of the node interactions in a linear time using hierarchical hash labels.

The label pair feature space of graph H for each label pair (l_i, l_j) is defined as follows [Zhang et al. 2011]:

$$\varphi_{l_i,l_j}(H) = \sum_{q=0}^{\infty} \tau_q |\{w \in W_q(H) : f_1(w) = l_i \wedge f_{q+1}(w) = l_j\}| \tag{2}$$

where $W_q(H)$ is the set of all possible walks with q edges in graph H, $f_1(w)$ is the first node of walk w, $f_{q+1}(w)$ is the last node of walk w, and $(\tau_0, \tau_1, \cdots, \tau_q)$ are weights of edges. Each edge is weighted by the summation of source and destination node colors (e.g., $c(i) + c(i+1)$, where function c calculates the node color, discussed in Section 3.3). In correspondence with feature map provided above, the graph kernel function based on label pairs is calculated as in [Zhang et al. 2011] as follows:

$$K(H, H^{'}) = \langle \varphi(H), \varphi(H^{'}) \rangle = \langle L(\sum_{i=0}^{\infty} \tau_i E^i) L^T, L^{'}(\sum_{j=0}^{\infty} \tau_j E^j) L^{'T} \rangle$$

$$= \sum_{m=0}^{|k|} \sum_{n=0}^{|k|} \left[L \left(\sum_{i=0}^{\infty} \tau_i E^i \right) L^T \right]_{mn} \left[L^{'} \left(\sum_{j=0}^{\infty} \tau_j E^j \right) L^{'T} \right]_{mn} \tag{3}$$

where E^i is the adjacency matrix of H, and L is the labeled matrix of H.

The manner by which the label process is made is described as follows. We denote a label as a binary vector $e = \{u_1, u_2, \ldots, u_r\}$ consisting of r-bits (0 or 1), representing the presence (1) of the group of instructions (discussed in Section 3.3) in a node. Let $XOR(e_i, e_j) = e_i \oplus e_j$ symbolise the XOR operation between two bit vectors of e_i and e_j. Let $ROT_o(e) = \{u_{o+1}, u_{o+2}, \ldots, u_r, u_1, \ldots, u_o\}$ denote the rotation (ROT_o) operation for $e = \{u_1, u_2, \ldots, u_r\}$, which shifts the last $r - o$ bits to the left by o bits and moves the first o bits to the right.

In order to compute the neighborhood hash of a graph, we first obtain the set of adjacent nodes $N^{adj}(n) = \{N_1^{adj}, \ldots, N_d^{adj}\}$ for each node n, and then calculate a neighborhood subgraph hash label for every node, using the following equation 4 [Zhang et al. 2012], where $l_i(n)$ indicates bit label of node n.

$$l_{i+1}(n) = NH(n) = ROT_1(l_i(n)) \oplus (ROT_o)(l_i(N_1^{adj})) \oplus \cdots \oplus ROT_o(l_i(N_d^{adj})) \tag{4}$$

As a way to differentiate between an outgoing and an ingoing edge, we set two ROT_o operations. If the edge $n_1 n$ is an incoming edge to node n, let $ROT_o = ROT_2$; if the edge nn_1 is an outgoing edge of node n, let $ROT_o = ROT_3$. It is worth nothing that $l_0(n)$ describes the information of node n, while $l_1(n)$ represents the label distribution of node n and its adjacent nodes. Finally, the structural information of subgraph of radius i is presented by $l_i(n)$. According to our experiments, we find a radius of $r = 2$ is the best choice for our system.

According to hierarchical hash labels, the graph kernel is defined as [Zhang et al. 2011]:

$$K(H, H^{'}) = \sum_{r=0}^{r^*} \beta r \left\langle Lr \sum_{i=0}^{\infty} \left(\tau_i E^i \right) (Lr)^T, Lr^{'} \sum_{j=0}^{\infty} \left(\tau_j E^j \right) (Lr)^{'T} \right\rangle$$

$$= \sum_{r=0}^{r^*} \sum_{m=0}^{|k|} \sum_{n=0}^{|k|} \beta r \left[Lr \left(\sum_{i=0}^{\infty} \tau_i E^i \right) Lr^T \right]_{mn} \left[Lr^{'} \left(\sum_{j=0}^{\infty} \tau_j E^j \right) Lr^{'T} \right]_{mn} \tag{5}$$

where E is the adjacency matrix of H and $L_0, L_1, \ldots, L_r$ are the hierarchical hash labels of H. For more details, we refer the reader to [Zhang et al. 2011; Zhang et al. 2012].

From a practical perspective, the whole process involved in calculating hierarchical hash labels is linear with respect to the size of the graph [Zhang et al. 2011; Zhang et al. 2012]. Consequently, computing the similarity between two control flow graphs will be equivalent to comparing the set of hash values.

3.4.3. z-score Calculation.

The last component concerns the distribution of opcode frequencies, since each set of opcodes that belong to a specific function will likely follow a specific distribution due to the functionality they implement. For this purpose, we utilize the z-score in order to convert these distributions into scores. In essence, a z-score $Z = \frac{(x-\mu)}{SD}$ indicates how many standard deviations an element is from the mean.

We calculate the z-score for each opcode distribution to facilitate accurate comparisons. Based on the possible values for the z-score, we obtain a curve distribution, where the area under the curve provides one feature value for each function. The area under the curve is calculated as $P(min{<}Z{<}max) = P(Z{<}max) - P(Z{>}min)$ [zta 2016].

3.4.4. Bayesian Network Model.

We use a Bayesian Network (BN) model for measuring the knowledge obtained from each component and for automating the interaction amongst these components. In addition, BN can depict the relations between the three proposed components, and would encode probabilistic relationships among their outputs. Moreover, situations where certain data for such components are not sufficient for identification can be handled by BN. A BN can be used to gain knowledge from a FOSS function identification problem domain as well as to predict the consequences of intervention, since it can be used to learn causal relationships. This feature is very important in the case of modifications performed by malware writers. Hence, the BN can capture both causal and probabilistic relationships, and is an ideal representation for combining prior knowledge and data.

As previously described, our system encompasses three main components to identify a FOSS function f: HMM (H), CFG-walks (W), and z-score (Z). Each component provides particular knowledge about the FOSS function, and the provided knowledge is measured by a factor Ψ_s. If the factor $\Psi_s \leq \Omega$, where Ω is a probability threshold value set by the Bayesian network, then our system is automatically transferred to another component, which means that the knowledge obtained from the current component is not sufficient. In addition, each component has a direct effect on the use of the other one; for instance, component H has a direct effect on component W. The situation can thus be modeled with a Bayesian network model. The joint probability function would be calculated as $P(f, H, W, Z) = P(f|H, W, Z)P(f|H, W)P(f|H)P(H)$, and the probability is defined with Bayes' law by equation 6:

$$p(y|\vec{x}) = \frac{p(y)p(\vec{x}|y)}{p(\vec{x})} \tag{6}$$

where $p\ (\vec{x}|y)$ is the probability of a possible input $x = (x_1, \ldots, x_n) \in \Upsilon^n$ given the output $y = (y_1, \ldots, y_n) \in \Gamma^n$. We define a set of conditional probabilities (factors) Ψ_1, Ψ_2, and Ψ_3 for our three components. Through extensive experiments applying logistic regression [Czepiel 2002], we found that, for our experimental settings, the best values for these factors are 0.45, 0.35, and 0.2, respectively. These factors are based on all possible features in the components, and thus represent more explicitly the underlying probability distribution of the features in each component. Each part of the joint probability is obtained by the equation 7.

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{p(x,y)}{\sum\limits_{y} p(x,y)} = \frac{\frac{1}{Z}\Pi_{s\in S}\Psi_s(x,y)}{\frac{1}{Z}\sum\limits_{y'}\Pi_{s\in S}\Psi_s(x_s,y_s)} = \frac{1}{Z(x)}\Pi_{s\in S}\Psi_s(x,y) \tag{7}$$

where Ψ_s represents the factor of the component $s = \{1,2,3\}$; $\prod$ represents the summation of the product of probabilities from each component; Z is the probability distribution [McDonald and Pereira 2005]; x is the set of features in each component; and y is the set of functions. We therefore obtain the following equation: $p(y,\vec{x}) = p(y)\frac{1}{Z(\vec{x})}\Pi_{s\in S}\Psi_s(\vec{x},y)$.

3.5. Implementation Environment

We develop a proof-of-concept implementation in python to evaluate our technique. The current version of our system works with 32-bit executables of both MS Windows (PE) and Linux (ELF). All our experiments are conducted on machines running Windows 7 and Ubuntu 15.04, with Core i7 3.4GHz CPU and 16GB RAM. The following steps are followed in the implementation. The binary programs are disassembled using IDA Pro Disassembler; however, if IDA Pro cannot find all of the code or determine the entry point, the Paradyn [Par 2016] tool is used. Function filtration is performed to drop standard library functions from which no features need to be extracted using our previous tools [Rahimian et al. 2015]. After the filtration process, the aforementioned features are extracted from FOSS packages, and then stored in a PostgreSQL database. Our dataset is composed of 160 projects, the details of which are described in Section 4.1. We additionally choose `Zeus`, `Citadel`, `Flame`, `Stuxnet`, and `Duqu` malware samples to demonstrate the identification of reused FOSS functions in malware binaries.

4. EVALUATION

In this section, in order to evaluate the effectiveness of our system, we test 160 real projects that reuse FOSS packages. We also select a number of modern malware binaries that are known to commonly reuse FOSS packages and evaluate *FOSSIL* against these binaries in Section 5. The performance criterion is the accuracy (F_2 measure) with which our system identifies the FOSS functions in malware binaries. In addition, we examines the effect of light obfuscation and the robustness of the system when it is confronted with different compilers and optimization levels.

4.1. Dataset Preparation

We manually gather a collection of FOSS packages from different sources and store them along with their features in a repository. Developing this repository is the first step towards the ultimate goal of building a large index of FOSS packages. To determine which FOSS packages are most widely incorporated, it is helpful to study code reuse on open repositories such as Github. The method for selecting the reused code involves assessing both the most popularly projects and the most reused libraries in modern malware based on the existing reports. After gathering the list of FOSS packages, we download their source code and compile them with Visual Studio (VS) 2010, VS 2012, and GNU Compiler Collection (GCC) 5.1 compilers. Furthermore, we obtain binaries and their PDBs from their official websites (e.g., WireShark); the compiler of these binaries are detected by a packer tool called ExeinfoPE [Exe 2016]. We evaluate our approach on a set of binaries, where a subset of them is detailed in Table III.

Table III: An excerpt of the selected FOSS packages

Project	Version	No. Fun.	Size(kb)	Project	Version	No. Fun.	Size(kb)
7zip/7z	15.14	133	1074	lshw	B.02.18	2090	2545
7zip/7z	15.11	133	1068	lzip	1.19	3341	1552
avgntopensslx	14.0.0.4576	3687	976	Mersenne Twister	1.10	321	2608
bzip2	1.0.5	63	40.0	miniz	2.8	327	121
expat	0.0.0.0	357	140	ncat	0.10rc3	462	373
firefox	44.0	173095	37887	Notepad++	6.8.8	7796	2015
fltk	1.3.2	7587	2833	Notepad++	6.8.7	7768	2009
FileZilla	3.27.0.1	97	7701	nspr	4.10.2.0	881	181
glew	1.5.1.0	563	306	nss	27.0.1.5156	5979	1745
Hasher	1.7.0	232	183	openssl	0.9.8	1376	415
hashdeep	4.3	3096	965	pcre3	3.9.0.0	52	48
info-zip/funzip	6.0	79	28	python	3.5.1	1538	28070
info-zip/zip	3.1	343	297	python	2.7.1	358	18200
info-zip/unzip	6.0	230	231	putty/putty	0.66 beta	1506	512
ibavcodec	11.10	719	99875	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	Qt5Core	2.0.1	17723	3987
libgd	1.3.0.27	883	497	SQLite	11.0.0.379	1252	307
libgmp	0.0.0.0	750	669	tinyXML	2.0.2	533	147
libjpeg	0.0.0.0	352	133	TestSSL	4	565	186
libpng	1.2.51	202	60	TrueCrypt	7.2	1193	2514
libpng	1.2.37	419	254	ultraVNC/vncviewer	1.2.13	4410	2045
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	xampp	5.6.15	5594	111436

4.2. Evaluation Metrics

Our ultimate goal is to discover as many relevant functions as possible with less concern about false positives, which means that recall has higher priority than precision. Hence, we choose to use the F_2 measure because it weights recall twice as heavily as it weights precision. The precision, recall, false positive rate (FPR), total accuracy (TA) and F_2-measure metrics are defined as follows [Davis and Goadrich 2006]:

$$Precision = \frac{TP}{TP+FP}, \;\; Recall = \frac{TP}{TP+FN}, \;\; FalsePositiveRate(FPR) = \frac{FP}{FP+TN}$$

$$TotalAccuracy(TA) = \frac{TP+TN}{TP+TN+FP+FN}, \;\; F_2 = 5.\frac{Precision.Recall}{4Precision+Recall} \tag{8}$$

where TP indicates number of relevant functions that are correctly retrieved; FN presents the number of relevant functions that are not detected; FP indicates the number of irrelevant functions that are incorrectly detected; and TN returns the number of irrelevant functions that are not detected.

In our experimental setup, we split the collected binaries into ten sets, reserving one as a testing set and using the remaining nine sets as the training set. We repeat this process 100 times and report the average output of the system in terms of aforementioned evaluation metrics. These metrics are calculated at function level (Section 4.3.1) and at project level (Section 4.3.2 and Section 4.4).

4.3. *FOSSIL* Accuracy

In this subsection, we test *FOSSIL* in the context of several scenarios: (i) examining the effect of Bayesian network model; (ii) examining accuracy across different versions of FOSS packages; and (iii) comparing *FOSSIL* with existing state-of-the-art solutions.

4.3.1. Effect of Bayesian network model.
We evaluate the accuracy of our system by examining it on a randomly collected binaries compiled with VS 2010, VS 2012, and GCC compilers from our repository. We test our system and report the precision, recall, and F_2 measure metrics. The results are summarized in Table IV, without and with the use of Bayesian network (BN) model.

Table IV: Effect of Bayesian network model

	Without a BN model			With a BN model		
Features	**Prec.**	**Rec.**	F_2	**Prec.**	**Rec.**	F_2
Opcodes	0.76	0.80	0.79	0.82	0.86	0.85
CFG-walks	0.72	0.76	0.75	0.84	0.83	0.83
Opcode distributions	0.70	0.72	0.71	0.81	0.86	0.85
Average / All together	0.727	0.76	0.75	0.93	0.84	0.86

4.3.2. Accuracy across different versions of FOSS packages.
We are further interested in evaluating *FOSSIL* with different versions of FOSS packages. For this purpose, we collect three different versions of all 160 projects in our repository, compile them with VS 2010 and test *FOSSIL*. The average output of the system in terms of precision, recall, and F_2 measure metrics is reported in Table V. The highest obtained F_2 measure is 0.863 which is related to `openssl`, and the lowest one is 0.727 for `lcms`. The low F_2 measure for `lcms` can be attributed to the presence of many small functions that have been inlined.

Table V: Accuracy results of different versions of FOSS packages

Project	**Prec.**	**Rec.**	F_2	**Project**	**Prec.**	**Rec.**	F_2
`SQLite`	0.78	0.81	0.803	`libxml2`	0.76	0.78	0.775
`Webph`	0.80	0.74	0.751	`libjsoncpp`	0.84	0.83	0.831
`Xterm`	0.79	0.81	0.805	`Mersenne Twister`	0.81	0.79	0.793
`Hashdeep`	0.81	0.85	0.841	`libssh2`	0.80	0.79	0.791
`TinyXML`	0.79	0.74	0.749	`openssl`	0.83	0.88	**0.863**
`libpng`	0.77	0.79	0.785	`bzip2`	0.79	0.80	0.797
`ultraVNC`	0.73	0.80	0.785	`UCL`	0.73	**0.9**	0.859
`lcms`	0.81	0.71	0.727	`TrueCrypt`	0.77	0.79	0.785
`libavcodec`	0.80	0.82	0.815	`liblivemedia`	0.80	0.81	0.807
`info-zip`	0.76	0.79	0.783	`Libavutil`	**0.84**	0.86	0.855
`Firefox`	0.77	0.81	0.802	`Expat XML parser`	0.80	0.8	0.8

Accuracy Interpretation. Our results demonstrate the following points:

(1) *Pre-processing*: Some of the top-ranked opcode features are related to the compiler functions (e.g., stack frame setup operations). It is thus necessary to filter out compiler functions to ensure better precision. Accordingly, in future work, we will leverage BINCOMP [Rahimian et al. 2015] with the current version of *FOSSIL* to distinguish compiler-related functions and FOSS-related functions. This will lead to considerable time savings and help shift the focus of the analysis to more relevant functions.
(2) *Project Type*: We found that the accuracy of *FOSSIL* depends on the type of projects. For instance, in our experiments, *FOSSIL* achieves high accuracy when it discovers cryptography libraries since these libraries generally have more arithmetic and logical operations. Also, *FOSSIL* is able to identify unique CFG-walks

that are related to certain cryptography operations. In contrast, we found that the accuracy of *FOSSIL* is slightly lower when it deals with parser libraries because they have functionalities in common with other libraries. For instance, `libucl` parser has common functionality with `JSON`; moreover, it can be integrated with a scripting language, such as `lua`. In Table V, reasonably good precision is observed for `openSSL`, while the precision for `libxml2` is 0.76. To tackle the effects of project type, we have to integrate more semantic features, e.g., type inference.

(3) *Project Size*: We observed through experiments that the accuracy of our system is not affected by the size of the function or of the project. For example, a comparison of the precision achieved by FOSSIL for `Firefox` and `openSSL` (0.77 and 0.83, respectively) with that of `libpng` and `bzip2` (0.77 and 0.79, respectively) illustrates that our features can be extracted regardless of the size of functions, and that they can reveal the semantics of any piece of code regardless of its size.

(4) *Features Extraction Level*. Typically, existing methods extract features from only one code level: instruction, function, or program level. A great advantage of FOSSIL is that it extracts features from all levels, making it possible to discover a function through different aspects. Also, the effect of code transformations such as the use of different compilers is reduced. In addition, we leverage concepts from biology to both extract the semantics of structural features and to improve efficiency when we deal with structural features.

(5) *Parameter Selection*. For the Bayesian network model, *FOSSIL* uses three parameters Ψ_1, Ψ_2, and Ψ_3, with values of 0.45, 0.35, and 0.2, respectively. Applying different values to the Bayesian network model makes it possible to achieve various trade-offs between precision and recall, as shown in Figure 5. Tuning these parameters may result in different values for precision and recall.

4.4. Comparison

We compare our system to existing state-of-the-art systems: IDA FLIRT [ida 2011], RENDEZVOUS, [Khoo et al. 2013], SARVAM [Nataraj et al. 2013], BINCLONE [Farhadi et al. 2014], TRACY [David and Yahav 2014; tra 2016], SIGMA [Alrabaee et al. 2015], and LIBV [Qiu et al. 2016]. Further details about these approaches can be found in Section 7. The code of all aforementioned systems are available, with the exception of RENDEZVOUS. We re-implement RENDEZVOUS with paying special attention to the definition of its characteristics as well as its stated assumptions.

It is worthy to note that since FLIRT is a signature-based technology, for the sake of comparison, it is required to create a set of signatures for the projects being evaluated. To this end, we employ FLAIR technology [FLA 2016], though the process is not fully automated and is considered a time-consuming task. Certain statistics regarding the FLIRT signatures generated by FLAIR are shown in Table VI. As can be seen from the table, the number of functions in the FOSS package corpus for which FLIRT had signatures is 457, which is approximately 14% of the total created signatures. This low percentage can be explained by the main goal of FLIRT technology, which is to identify the standard library functions such as C-standard libraries. In addition, the percentage of signature collision is 19%, which must be fixed by extending the signature formed; this further increases time consumption.

Since each of the existing systems use different metrics to measure the accuracy, we unify the metric by using precision, recall, total accuracy (TA), and false positive rate (FPR). The obtained accuracy results on some projects as well as ROC curve of all projects are shown in Table VII and Figure 3, respectively.

As can be seen, our system effectively identifies FOSS functions in the selected projects, and returns an average of 95% precision and 89% recall. Its accuracy is su-

Table VI: Statistics about FLIRT signatures on the FOSS packages

Category	**No. of Signatures**	**Example**
Compression	300	E.g., Zlib, Bzip, UCL, infozip
Encryption	313	E.g., Botan, OpenSSL, TrueCrypt
Graphics	351	E.g., bgfx, openVDB, libpng
Web browser	307	E.g., crow, libOnion, firefox
Parsing	280	E.g., Expat, LibXml, TinyXml
Multimedia	171	E.g., LibVLC, SDL,
Database	178	E.g, MySQL++, SQLite, LMDB++, redis3m
JSON	204	E.g., json, jbson, libjson, jsonCPP
Networking	591	E.g., Restbed, Libcurl, Putty, WebSocket
Scripting	222	E.g., glew, lua
Math	70	E.g., libgmp
Editors	76	E.g., Notepad++
Hashing	152	E.g., Hashdeep, pHash, blockhash
Total	**3215**	

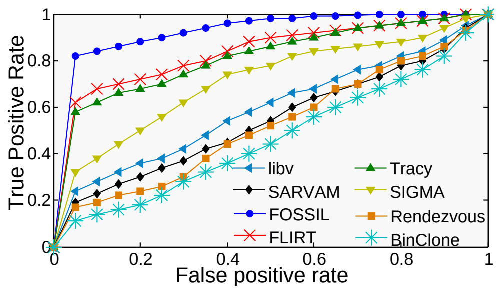


Fig. 3: ROC curve

perior to that of the other systems, including FLIRT technology, which achieves the second highest rate of average precision 91% and 78% recall, as well as TRACY, which yields in some projects the highest rate of precision of 82%. On the other hand, BINCLONE achieves the lowest true positive rate since it employs exact matching, which causes a high rate of false positives. The reason of increase in precision rate in FOSSIL could be because of the combination types of the features (e.g., semantic and syntactic). In addition, ranking process helps to reduce the general and irrelevant opcodes in order to increase the accuracy. Moreover, our system employs Bayesian network model that can control false positives rates by defining three threshold values.

An analysis of other systems reveals various limitations. TRACY is more accurate compared with the other systems, since data flow constraints are applied on `tracelets` (decomposing CFGs into subtraces of fixed length, excluding jump instructions). However, TRACY assumes that the candidate function should contain at least 100 basic blocks [David and Yahav 2014]; otherwise, it has a high rate of false positives. SIGMA integrates different graph representations, such as register flow graph, control flow graph, and call graph, to represent more semantics, whereas the approach is computationally expensive. The features used by RENDEZVOUS, such as `n-grams` and `k-CFGs`, are sensitive to code changes that lead to more false positive rates. Although grayscale images used by SARVAM are rich sources of information, they include many irrelevant features that increase the rate of false positives. LIBV generates execution dependence graphs (EDG) by applying data and control flow constraints; however, some issues such as having isomorphic EDGs for two different functions affect the accuracy.

Table VII: Accuracy results of different existing approaches

		FireFox	Zlib	Jsoncpp	Libpng	OpenSSL	Python	Wireshark	Curl	TinyXML	Xaamp
TA	FLIRT	0.56	0.66	0.74	0.75	0.83	0.84	0.70	0.82	0.76	0.76
	Tracy	0.55	0.66	0.77	0.65	0.73	0.82	0.76	0.61	0.50	0.60
	SIGMA	0.65	0.69	0.74	0.70	0.71	0.82	0.76	0.62	0.48	0.58
	BinClone	0.49	0.63	0.63	0.64	0.66	0.69	0.76	0.58	0.42	0.60
	Rendezvous	0.53	0.65	0.64	0.65	0.63	0.68	0.65	0.53	0.39	0.52
	SARVAM	0.69	0.74	0.76	0.69	0.73	0.84	0.75	0.66	0.51	0.6
	libv	0.49	0.57	0.66	0.62	0.69	0.72	0.77	0.62	0.42	0.60
	FOSSIL	0.98	0.80	0.93	0.84	0.90	0.85	0.90	0.87	0.81	0.79
FPR	FLIRT	0.09	0.26	0.13	0.18	0.49	0.32	0.26	0.34	0.26	0.39
	Tracy	0.35	0.51	0.36	0.22	0.48	0.51	0.35	0.52	0.25	0.46
	SIGMA	0.36	0.53	0.36	0.24	0.49	0.51	0.37	0.52	0.25	0.46
	BinClone	0.36	0.67	0.51	0.27	0.54	0.45	0.41	0.55	0.45	0.46
	Rendezvous	0.37	0.61	0.43	0.27	0.52	0.44	0.53	0.55	0.45	0.46
	SARVAM	0.39	0.53	0.35	0.20	0.49	0.51	0.38	0.52	0.21	0.48
	libv	0.49	0.54	0.46	0.31	0.44	0.42	0.45	0.48	0.50	0.56
	FOSSIL	0.15	0.28	0.25	0.19	0.19	0.26	0.39	0.38	0.18	0.39
Prec.	FLIRT	0.93	0.88	0.95	0.93	0.90	0.94	0.89	0.93	0.92	0.88
	Tracy	0.77	0.78	0.90	0.86	0.84	0.89	0.90	0.72	0.77	0.74
	SIGMA	0.84	0.79	0.89	0.88	0.83	0.89	0.88	0.73	0.75	0.72
	BinClone	0.73	0.70	0.76	0.84	0.77	0.81	0.87	0.69	0.55	0.74
	Rendezvous	0.75	0.72	0.79	0.85	0.75	0.80	0.75	0.64	0.5	0.66
	SARVAM	0.85	0.82	0.89	0.89	0.84	0.90	0.87	0.76	0.79	0.73
	libv	0.66	0.68	0.80	0.81	0.81	0.84	0.87	0.73	0.52	0.70
	FOSSIL	0.99	0.93	0.98	0.96	0.98	0.95	0.96	0.94	0.95	0.90
Rec.	FLIRT	0.58	0.72	0.75	0.76	0.90	0.88	0.73	0.87	0.8	0.82
	Tracy	0.65	0.78	0.83	0.64	0.83	0.90	0.81	0.74	0.53	0.71
	SIGMA	0.74	0.81	0.80	0.69	0.82	0.90	0.82	0.75	0.51	0.69
	BinClone	0.60	0.81	0.77	0.66	0.80	0.78	0.82	0.73	0.50	0.70
	Rendezvous	0.63	0.80	0.74	0.67	0.77	0.76	0.77	0.68	0.45	0.62
	SARVAM	0.78	0.84	0.81	0.67	0.83	0.91	0.81	0.78	0.50	0.71
	libv	0.65	0.72	0.77	0.65	0.77	0.79	0.85	0.72	0.52	0.75
	FOSSIL	0.99	0.84	0.94	0.85	0.91	0.88	0.93	0.90	0.82	0.85

Note: We use following abbreviations: total accuracy (TA), false positive rate (FPR), precision (Prec.), and recall (Rec.).

Performance. We also compare the performance of each system by computing the overall execution time, which involves the feature extraction, and searching through the repository to find matches. The purpose of measuring performance is to evaluate the practicality of each system for large-scale datasets. For this purpose, no time limit is set to finish the FOSS function identification. However, we notice that some approaches such as BinClone, SIGMA, and libv are taking long time to detect functions since they are not scalable enough to obtain the search result within a specific given time frame.

In particular, the execution time for FOSS function identification in *FOSSIL* was measured by adding the time required for each step (normalization, opcode ranking, and feature extraction in each component) to the time spent to discover the FOSS functions. Feature extraction in the first component takes 5 sec for the small packages in our dataset (e.g., 100 functions) and 15 sec for the large package (e.g., 50,000 functions). The proposed hash subgraph kernel is fast, taking an average of 5 sec for all packages in a similar environment. The time required to extract features in the third component is negligible (less than 1 ms). Our system spends the majority of time on searching the repository; further optimizing the search using advanced indexing techniques is a future direction. Each search iteration takes a minimum of 7 sec and a maximum of 50 sec.

The overall time for FOSSIL ranges from 17 to 80 sec, while the averages for RENDEZVOUS, TRACY, SARVAM, SIGMA, and LIBV are 72.5, 115, 55, 155, and 111.5 sec, respectively. We observe that the performance of SARVAM is closer to that of *FOSSIL*, since the extraction of image features is relatively efficient. The performance of RENDEZVOUS is also close since it uses a Bloom filter, which speeds up the retrieval process.

4.5. Scalability study

Since one of our ultimate goals is to build a searchable index for large-scale FOSS projects based on the proposed approach in this paper, in addition to time efficiency, we evaluate the scalability of FOSSIL when it is used to index and retrieve matched functions on a large number of projects. This made it possible to investigate the trade-off between accuracy and efficiency. For this purpose, we add more projects, dlls, operating system applications, and other programs to our repository. In total, there are 500 applications and approximately 1.5 million functions. We measure the total time required to index the project and to match the target files. In addition, we examine the accuracy of each component separately and all together. Figure 4 shows that our system is scalable when the number of functions reaches to 1.5 million.

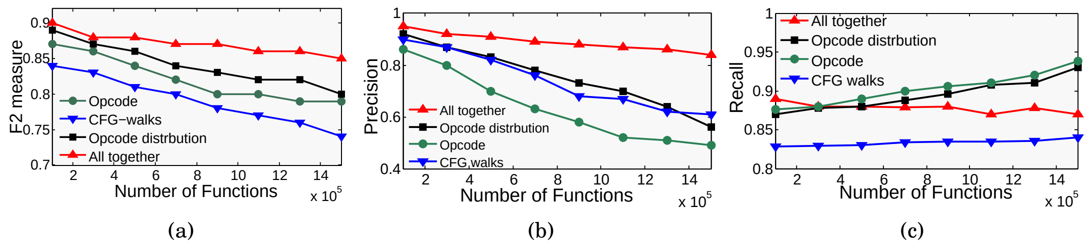


Fig. 4: The performance of *FOSSIL* against a large set of functions

The F_2 measure falls down slightly, from 0.9 to 0.86, which provides some insight into the scalability of system when it deals with a large number of FOSS functions. Based on these results, we believe our system will be efficient and practical for most real-world applications.

4.6. The confidence estimation of a Bayesian network

Using a Bayesian network model provides a confidence estimator based on probability scores, where higher probability scores correspond to higher confidence. Future research will hopefully produce an actual probability score. Applying different factor values to the Bayesian network model makes it possible to achieve various trade-offs between precision and recall. Figure 5 shows the results of confidence estimation for three factors, varying the trade-off between precision and recall. A precision measure of 50% is achieved with a recall measure of just under 80%; conversely, 50% recall gives over 80% precision.

4.7. The impact of evading techniques

We consider the projects from our dataset in order to test FOSSIL against binary and source obfuscation, as shown in Table VIII. The obfuscation process is done in two stages. First, C^{++} refactoring tools [ref 2016b; 2016a] are used for source code level

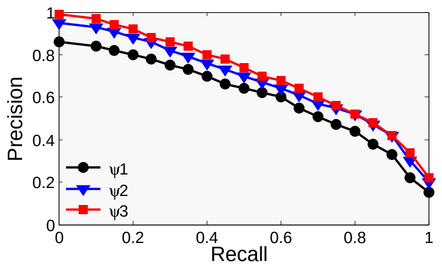


Fig. 5: Confidence estimation: precision vs. recall

obfuscation. These rely on the following techniques: *moving a method from a superclass to its subclasses*, and *extracting a few statements and placing them in a new method*. We refer the reader to [Fowler 1999] for in-depth explanations of these techniques. We also apply Nynaeve [nyn 2016] tool, which comprises *Frame Pointer Omission* and *Function inlining* methods.

Table VIII: Evading technique tools, methods, and their effects on FOSSIL components.

Tool	Method	Input	Output	A^*	Component		
					Opcode	CFG-Walk	Opcode Dist.
LLVM [Junod et al. 2015a]	CFG flattening	Bin	Bin	74%	○	●	○
	Instruction substitution			84%	●	○	○
	CFG bogus			81%	○	●	○
DaLin [Lin and Stamp 2011]	Instruction reordering	Asm	Asm	86%	○	○	○
	Dead code insertion			86%	○	○	○
	Register renaming			86%	○	○	○
	Instruction substitution			84%	●	○	○
Trigress [tig 2016]	Virtulazation	Src	Src	82%	●	●	●
	Jitting			83%	●	○	○
	Dynamic			83%	●	○	○
PElock [pel 2016]	Hide procedure call	Asm	Asm	86%	○	○	○
	Insert fake instruction			86%	○	○	○
	Prefix junk opcode			86%	○	○	○
	Insert junk handlers			86%	○	○	○
Nynaeve [nyn 2016]	Frame pointer omission	Src	Bin	81%	●	●	○
	Function inlining			80%	○	●	○
OREANS [ore 2016]	Binary encryption	Src	Bin	NA	NA	NA	NA
Gas Obfuscator [GAS 2016]	Junk byte	Asm	Asm	86%	○	○	○
Designed Script	Loop unrolling	Src	Src	77%	●	○	●

Note: (A^*) indicates accuracy after applying obfuscation method while the accuracy before applying obfuscation method is 86%. (○) indicates there is no effect, while (●) means the corresponding component get affected. (*NA*) means not applicable. We use the following abbreviations: (Bin) Binary, (Asm) Assembly, (Src) Source, and (Dist.) Distribution.

Second, to investigate binary-level obfuscation, we compile the 160 projects with GCC and VS compilers, and the resulting binaries are obfuscated using DaLin [Lin and Stamp 2011] generator and Obfuscator-LLVM [Junod et al. 2015b]. These obfuscators replace instructions with other semantically equivalent instructions (*instruction* *substitution*). Obfuscator-LLVM also applies *control flow flattening*, and *bogus control flow* techniques, whereas DaLin performs *instruction reordering*, *dead code insertion*, and *register renaming* as well. The obfuscated binaries are passed as target binaries to our system, and we then measure the accuracy of function identification.

Our system obtains an average F_2 measure of 83.1% in identifying similar FOSS functions, which represents only a slight drop in comparison to the 86% observed without obfuscation.

As can be seen in Table VIII, the obfuscation tools work at three levels: source, binary, and assembly. It can be observed that the obfuscation methods of *CFG flattening*, *function inlining*, and *loop unrolling* decrease the accuracy of FOSSIL by approximately $6 - 12\%$. However, their effect on accuracy is not significant since FOSSIL employs a Bayesian network to synthesize the knowledge obtained from the three components by defining a confidence estimator function. Table VIII also shows that FOSSIL cannot deal with encrypted binaries. The current version of FOSSIL consists of components relying on static analysis. A possibility for future work is to extend FOSSIL by including dynamic components that can deal with encrypted binaries.

There are three main reasons for the slight drop in accuracy. The first component, HMM, deals with opcode frequencies at the function level, so in the case of *instruction reordering*, all the reordered instructions, regardless of order, will be captured. In addition, since the operands are not considered in this component, *register renaming* does not affect the accuracy. However, this component is affected slightly by *instruction replacement* because this technique affects frequencies. However, as previously mentioned, the chi-squared test is used to evaluate the frequencies, and it involves a confidence interval that varies according to user requirements. The second component, CFG-walk, tolerates instruction-level obfuscation to a greater extent since it deals with the semantics of a function as well as the instruction groups. To avoid *bogus control flow* and *function inlining* techniques, we use the most important opcodes to color CFG-walks. We also label a node with its neighbors in a novel way in order to avoid any obfuscation that can affect the CFG. However, this component is affected by *CFG flattening*. The third component, z-score, measures the area of the opcode distribution, so both *instruction replacement* and *dead code insertion* may slightly affect it. In general, using opcode ranking, normalization, and coloring techniques reduce the effects of most aforementioned obfuscation methods. However, the Bayesian network model synthesizes the knowledge obtained from the three components; therefore, if the knowledge from one is not sufficient, the Bayesian network model will automatically assign more weight to the other components.

The Impact of Compilers. To create an experimental dataset, we consider 160 projects compiled with Visual Studio (VS), GNU Compiler Collection (GCC), Intel C++ Compiler (ICC), and Clang compilers with Od optimization setting. To measure the effect of different compilation options such as compiler optimization flags, we additionally compile them with level-1, level-2, and level-3 optimizations, namely the $O0$, $O2$, and Ox flags. We extract features for each compilation setting, and then test our system. The results illustrated in Table IX show that the features extracted by our system are greatly effective for most optimization speed levels.

The normalization process used in our system can reduce the effect of GCC and VS compilers. Moreover, the top-ranked opcodes are more related to the semantics of the function, in addition to the colored CFG-walks which help to avoid compiler effects. However, the accuracy drops significantly when the source compilers are Clang or ICC, since these compilers produce more variable code compared to VS and GCC compilers. Such limitations can be handled by first identifying the compiler using existing tools such as Exeinfo [Exe 2016] and then applying the suitable features accordingly.

Table IX: FOSS function identification with different compilers and compilation settings.

Compiler	Optimization Speed	Precision	Recall
VS	$O0$, $O2$, Ox	0.95, 0.95, 0.95	0.94, 0.92, 0.92
GCC	$O0$, $O2$, Ox	0.92, 0.92, 0.92	0.93, 0.90, 0.89
ICC	$O0$, $O2$, Ox	0.78, 0.74, 0.69	0.81, 0.80, 0.78
Clang	$O0$, $O2$, Ox	0.65, 0.59, 0.60	0.64, 0.60, 0.58

5. APPLYING *FOSSIL* TO REAL MALWARE BINARIES

We next test *FOSSIL* in determining the FOSS functions in Malware binaries. Due to the lack of ground truth, we test *FOSSIL* on two specific sets of malware binaries. The first set is consists of malware samples, which are analyzed by reverse engineers and security analysts to provide the possibility to match our findings with technical reports. The second set contains 12 malware families with 5000 samples; with this large set, we aim to study the stability and scalability of *FOSSIL*, and to derive insights from these malware samples by providing information about the type and percentage of reused FOSS packages.

5.1. Malware Dataset Analyzed by Technical Reports

In this set, we apply *FOSSIL* to the following malware binaries: `Zeus`, `Citadel`, `Stuxnet`, `Flame`, and `Duqu`. These samples are unpacked and de-obfuscated at our security laboratory.

5.1.1. Statistics of Our Findings.

Statistics regarding the number of identified FOSS functions by *FOSSIL* in malware binaries are shown in Table X. Our system is able to determine 645 FOSS functions in `Duqu`, 275 in `Stuxnet`, 298 in `Zeus`, 493 in `Flame`, and 500 in `Citadel`, where these functions belong to various FOSS packages in our repository. Our system established 87 matches between `Zeus` and *webph*; however, no matches were found between `Zeus` and *SQLite*, *hashdeep*, *TinyXML*, *libpng*, and *libavcodec* packages. In addition, the results show similarities between pairs of malware binaries in terms of reused FOSS functions. For instance, we found 55% of similarity in the use of FOSS functions between `Zeus` and `Citadel`, whose common functions are 80 *webph*, 35 *xterm*, 81 *ultraVNC*, 20 *lcms*, and 60 *info-zip* functions. *FOSSIL* found a degree of similarity of 38% among `Stuxnet`, `Flame`, and `Duqu`.

Table X: The number of specific FOSS functions with their percentages found in all variants of malware binaries.

	`Zeus`	`Citadel`	`Stuxnet`	`Flame`	`Duqu`	`Zeus`	`Citadel`	`Stuxnet`	`Flame`	`Duqu`
SQLite	-	15	175	285	373	0	2%	30%	36%	34%
webph	87	134	-	5	17	13%	15%	0	0.6%	1.5%
xterm	42	86	-	15	20	6%	10%	0	2%	2%
hashdeep	-	5	-	25	90	0	0.5%	0	3%	8%
TinyXML	-	10	35	70	60	0	1%	6%	9%	5%
libpng	-	-	30	20	60	0	0	5%	3%	5%
ultraVNC	86	125	-	-	-	13%	14%	0	0	0
lcms	20	25	-	-	-	3%	3%	0	0	0
libavcodec	-	-	25	40	10	0	0	4%	5%	1%
info-zip	63	100	10	33	15	9%	11%	2%	4%	1%

We analyze identified FOSS functions listed in Table X by studying their general functionality to obtain the percentage of false positive rates (FPR) for each category. The details of which are listed in Table XI. For instance, we find amongst 25 functions of *libavcodec* that are identified in `Stuxnet`, some perform information compression, and others are related to standards for digital audio, and video streams. The false positive rate is 4% which is out of 14% (the total false positive rate among all malware families).

The HMM component identifies more functions in malware binaries as it deals with opcode frequencies. As explained previously, we apply a chi-squared test to these frequencies. Subsequently, the second component extracts CFG-walks from the functions that pass the HMM. For this reason, the number of CFG-walk matches is less than or equal to the number of matches in the first component.

Table XI: Functionality of detected FOSS packages and obtained FPR

Category	**Functionalities**	**Example**	**FPR**
Web application	Send/receive information to/from bot server	*webph*	11%
Emulator	Keyboard symbols-to-unicode encoding at Terminal emulators	*xterm*	3%
Remote access	Accessing other machine remotely	*ultraVNC*	1%
Compression	Compression data	*info-zip*	0.5%
Video	Encoding/decoding audio, and video streams	*libavcodec*	14%
Network graphics	Handling PNG images	*libpng*	11%
Hashing	Computing hashes	*hashdeep*	10%
XML parser	Parsing XML files	*TinyXML*	12%
Database	Saving and retrieving information	*SQLite*	8%

Table XII summarizes the number of concrete features extracted from $60,000$ functions in malware samples, as well as the code property in which can be captured by that feature. The interaction properties are considered at the instruction level and at the control flow level, and at the function level.

Table XII: The number of concrete features in malware binaries and corresponding code properties (instruction-level, control-flow-level, and function level)

		Code Property		
Features	**#**	**Inst.**	**Control flow**	**Func.**
Opcode	132,067	*		
CFG-walks	280,869		*	
Opcode Distribution	99,098	*	*	*

5.1.2. Significance of the Proposed Features.
As illustrated in Table X, our tool discovers various number of FOSS functions in our malware dataset. In this part, additionally we examine the impact of each feature as well as the use of Bayesian network model on the number of discovered FOSS functions. Figure 6 shows the results obtained without a Bayesian network model. As can be seen, opcodes always discover more functions than other features. For instance, 440 functions related to `SQLite` package are discovered by opcode features, while CFG-walks and opcode distribution features identify 320 and 165 functions, respectively.

When the Bayesian network is included, in each level some functions are filtered according to a threshold value, leading to lower numbers of FOSS functions discovered,

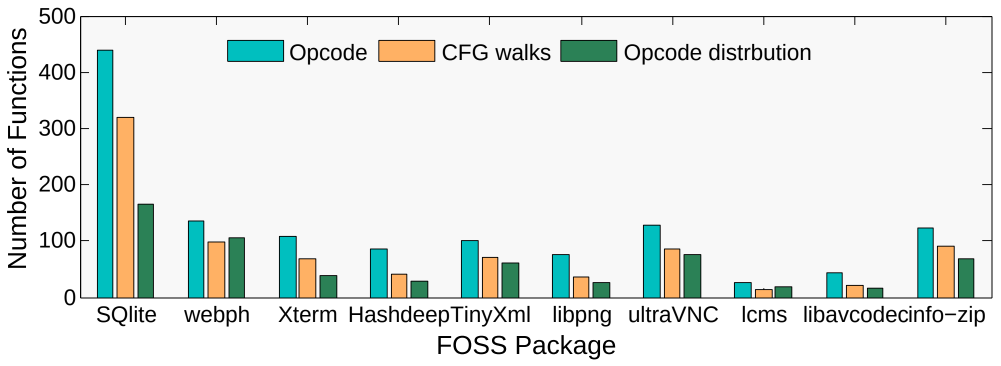


Fig. 6: The number of FOSS functions discovered by the proposed features without a Bayesian network

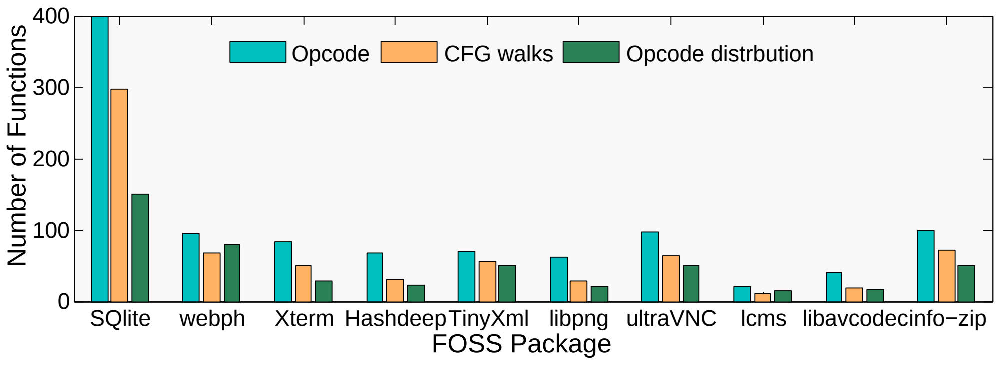


Fig. 7: The number of FOSS functions discovered by the proposed features using a Bayesian network

as shown in Figure 7. Consequently, based on the experiments, the number of false positive rates is reduced as well. For instance, 400 functions related to SQLite package are discovered by opcode features, while CFG walks and opcode distribution features identify 298 and 150 functions, respectively.

For each malware family, we also analyze the percentage breakdown of FOSS functions, compiler functions, and standard library functions. We apply BINCOMP [Rahimian et al. 2015] to determine the percentage of compiler-related functions, while we employ BINSHAPE [Shirani et al. 2017] to label standard library functions. For the sake of clarity, we categorize each FOSS package according to its main functionality, such as hashing, parsers, emulators, and compression. As shown in Figure 8, the percentages of FOSS functions, compiler-related functions, and standard library functions in `Zeus` are 41%, 25%, and 34%, respectively. Moreover, the percentages of reused FOSS functions in `Citadel` is close to that of `Zeus`, e.g., 39%. In contrast, the `Stuxnet` family (`Stuxnet`, `Flame`, and `Duqu`) exhibit high percentages of FOSS functions; we attribute this to the fact that these samples are designed to target control systems, which require the use of many libraries.

5.1.3. Comparison with technical reports.
In this section, we compare our results with the findings in existing technical reports

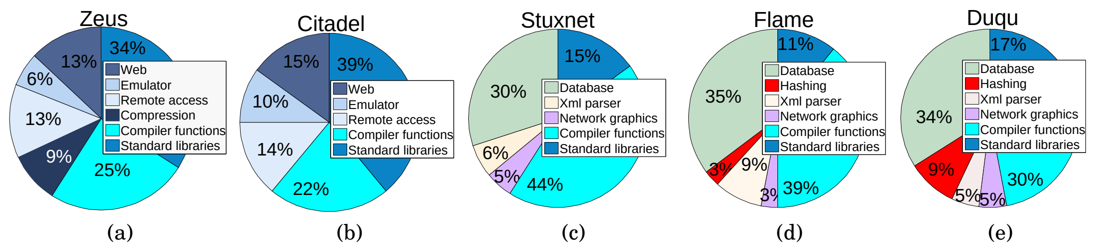


Fig. 8: Percentages of various types of functions found in malware families

as shown in Table XIII. It is worth noting that these technical reports do not reveal any statistics (e.g., number of functions) about their findings. This makes the verification process challenging. To verify our numbers, we follow the models in [Calvet et al. 2012; Krka et al. 2010] execute the functions and then compare execute traces in order to say if the functions are similar. We have found 10% of false positive rate. However, after we apply our valdiator (Bayesian network) to the results, the number of false positives has been reduced to 1.9%. It is worthy to mention that *FOSSIL* was able to detect functions in `Stuxnet` malware family that are related to following FOSS packages: `webph, xterm, and libavcodec`. These FOSS packaged were not identified by technical reports [GRe 2012; Boldizsr 2012]. To verify if these functions are correctly identified, we employ the dynamic approach in [Su et al. 2016]. Simply, we excute the matched function with the corresponding one in our repository and if the outputs are the same, we mark it as a corrected match. We found that 18% of these functions are false positives while 82% are true positives.

Table XIII: Matching our findings with existing technical report findings

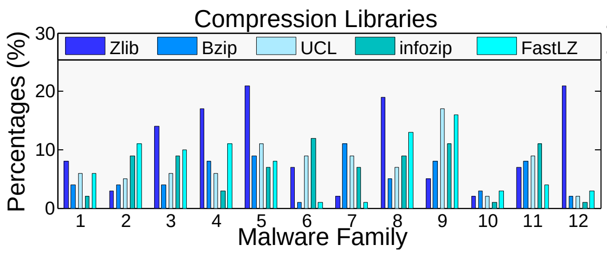
Malware	FOSS Packages	Technical Reports			
		[GRe 2012]	[Boldizsr 2012]	[Khoo 2013]	[Milletary 2012]
`Zeus/` `Citadel`	`SQLite, webph, xterm`			✗, ✗, ✓	✗, ✓,✗
	`Hashdeep, TinyXML, ultraVNC`	*NA*	*NA*	✗, ✗, ✓	✓, ✗, ✓
	`lcms, info-zip`			✓, ✓	✓, ✓
`Stuxnet/` `Flame/` `Duqu`	`SQLite, webph, xterm`	✓, ✗, ✗	✓, ✗, ✗		
	`Hashdeep, TinyXML, libpng`	✗, ✓, ✓	✓, ✗, ✓	*NA*	*NA*
	`libavcodec, info-zip`	✗, ✓	✗, ✓		

Note: We use the following notations. (*NA*) The technical report does not mention corresponding malware. (✓) Our finding matches the finding obtained in the technical report. (✗) Our finding is not obtained in the technical report.

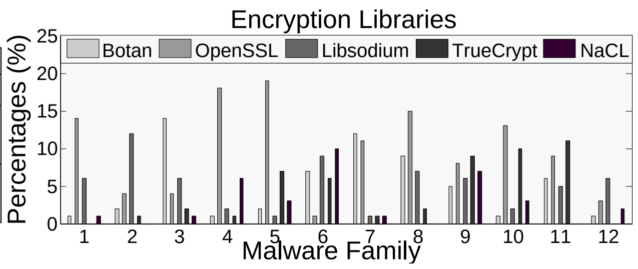
5.2. General Malware Dataset

We further apply *FOSSIL* to a large set of malware binaries with their variants, which are unpacked and de-obfuscated at our security laboratory. These samples are obtained from various resources as follows. In our security laboratory (Concordia Security Laboratory), we have very large datasets that have been collected since 2000 from our collaborators and from VirusTotal since 2004. We have also downloaded large dataset since three years from various websites including: VirusShare, Kaggle Microsoft Malware Classification Challenge, various malware farms such as KernelMode.info and contagion.org. We get part of the dataset used by [Jang et al. 2013], which is collected by the DARPA Cyber Genome program [DAR 2016].

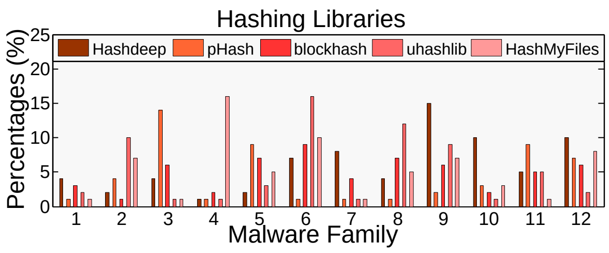
We choose a set of FOSS packages and provide their percentage of usages by certain malware families. There are several goals of doing this experiment. The first goal



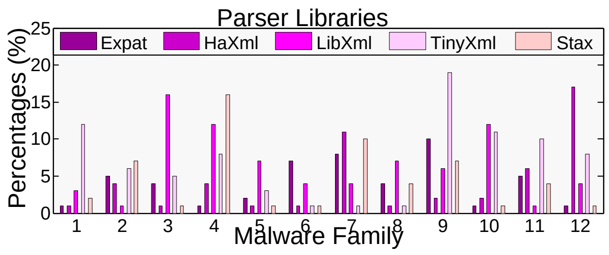
(a)



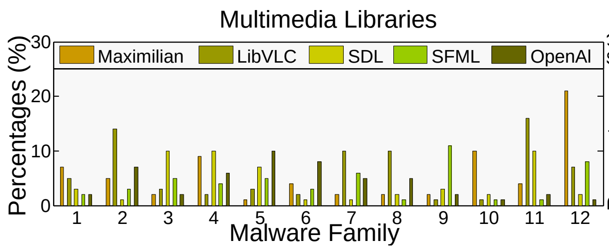
(b)



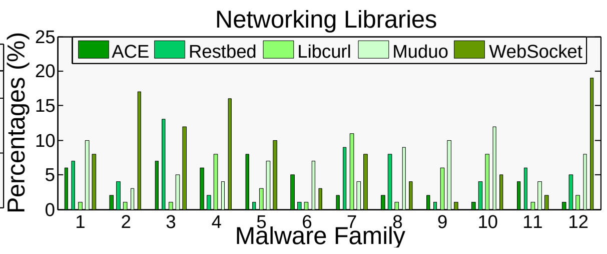
(c)



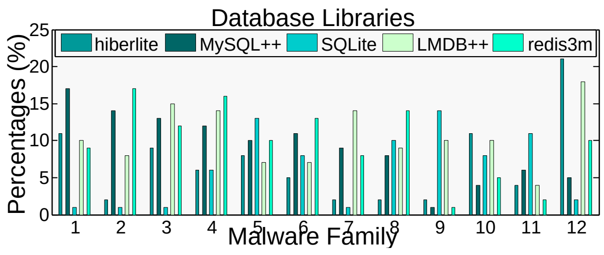
(d)



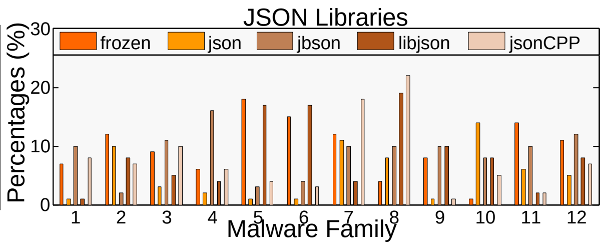
(e)



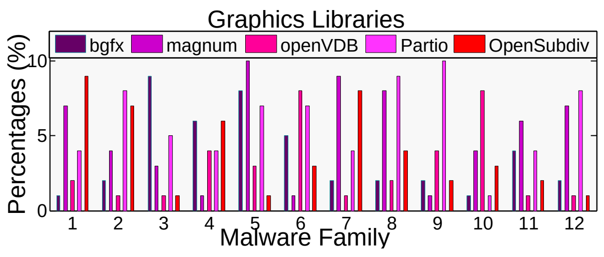
(f)



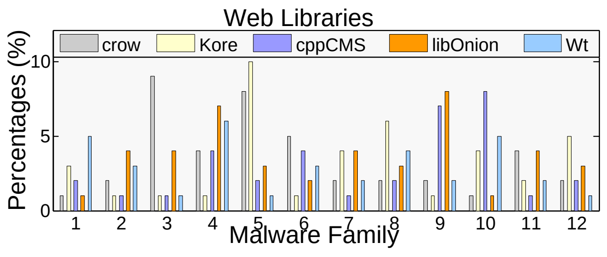
(g)



(h)



(i)



(j)

Fig. 9: FOSS packages found in certain malware. Numbers on x-axis represent (1) Ramnit (2) Lollipop (3) Kelihos (4) Vundo (5) Simda (6) Tracur (7) Gatak (8) Locky (9) Kovter (10) Samas (11) Drixed (12) Ursnif malware families.

involves providing clues about malware functionality that can help reverse engineers and security analysts gain a better understanding of the malware. The second goal involves determining the relation between FOSS packages and malware binaries from the same family by providing statistics about FOSS packages found in each malware binaries and other family members. Additionally, we study the relation between the use of FOSS packages and malware evolution. The goal is to verify whether FOSS usage is started in recent years or at the first version of the same malware.

5.2.1. Percentages of Reused FOSS Packages in Malware Families..
We have applied *FOSSIL* to almost 500 GB of malware samples provided by the Kaggle Microsoft Malware Classification Challenge (BIG 2015). These samples include bots, worms, and Trojan horses, including *KBot*, *BlasterWorm*, *MiniPanzer.A*, *CleanRoom.A*, *CleanRoom.B*, *MiniPanzer.B*, *CleanRoom.C*, *NBOT*, *Bunny*, *Casper*, *Kelihos_ver1*, *Kelihos_ver2*, and *Kelihos_ver3*. The results are illustrated in Figure 9. *FOSSIL* is able to find the following FOSS packages: i) compression libraries with an average percentage of 17%, where *zlib* is one of the most commonly used packages in malware binaries; ii) the encryption libraries with an average percentage of 6%. The reason for this low percentage is that most malware writers use their own customized encryption algorithm; iii) the percentages for hashing libraries, parser libraries, multimedia libraries, networking libraries, database libraries, JSON libraries, graphics libraries, and web libraries are 11%, 8%, 5%, 17%, 20%, 10%, 7%, and 9%, respectively.

5.2.2. Relation Between Reused FOSS Packages and Malware Evolution.
We are interested in determining the relation between use of FOSS packages and malware lineage. More specifically, we would like to verify whether a FOSS package is used over many years in the same series of a malware family. For this purpose, we use 114 samples with known evolution collected by Cyber Genome program [DAR 2016]. We show the relation between FOSS packages and the year of malware evolution in Figure 10. As seen, reusing FOSS packages from 2000 to 2006 is almost close to 0%, while it increases to 10% in 2008; starting from 2010, the percentage of using FOSS packages increases significantly. This observation supports the fact that modern malware borrow FOSS libraries. In addition, we notice that certain malware families use the same percentage of FOSS, which indicates that the functionality of that family has not been changed over time, whereas other malware families began to change their FOSS package due to their goals and functionalities.

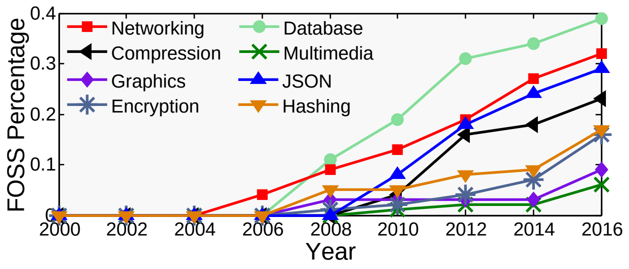


Fig. 10: The relationship between FOSS packages and the year of malware evolution

6. DISCUSSION

Obfuscation. We evaluate *FOSSIL* on benign and un-obfuscated binaries as well as against simple obfuscation, refactoring methods, and different compilers. We believe that our static approach could be applied to unpacked malware or to packed malware in combination with a universal unpacker [Martignoni et al. 2007; Kang et al. 2007]. Nevertheless, if a malicious author wishes to bypass our approach, he/she can obfuscate the binary; for example, by CFG flattening and loop unrolling. Other properties such as function behavior are more difficult, but still possible, to be obfuscated.

Assigning more semantics to *FOSSIL* outcome. The current version of our approach identifies functions that belong to FOSS packages, but does not annotate each function to the source repository (e.g., source to binary matching), which would improve program understanding. It would be possible to leverage prior work on source-assembly matching [Saxe et al. 2014; Rahimian et al. 2012]. We will also consider leveraging prior work on type inference [Lin et al. 2010; Lee et al. 2011] to assign additional semantics to our matching.

Improving accuracy. For the sake of efficiency, we have focused on features extracted statically from the executable. However, features extracted from program executions could also be incorporated into the *FOSSIL* system in order to improve accuracy. For example, functions that repeatedly use the same memory access to heap-allocated data structures may provide useful information.

Other applications. In this paper, we have demonstrated how our system discovers FOSS functions in binaries. We have also applied it to malware binaries. It is worthy to note that other applications for our system exist; for example, the features used by *FOSSIL* may be used to provide clues about vulnerabilities. This can be done by profiling a vulnerability as a feature in *FOSSIL*; then, once a target binary is given, *FOSSIL* can search for similar patterns. In addition, *FOSSIL* can be used as a tool for highlighting certain types of binary parts, such as compiler-related, FOSS-related, user-related, etc. *FOSSIL* could also be applied for malware triage. For instance, *FOSSIL* can be used first to identify which FOSS package is used; then, according to the importance of that package, the reverse engineer or security analyst can determine the binary with which he/she should begin.

7. RELATED WORK

Malware binaries are typically available resources that can aid in malware detection. These binaries can lead to various clues about malware; Kaspersky's researchers provide a comprehensive study of `Stuxnet` and `Duqu` malware and they state that these two malware were developed by the same malware environment [Kas 2016]. Another forensic analysis performed by FireEye [Moran and Bennett 2013] stated that "many seemingly unrelated cyber-attacks may, in fact, be part of a broader offensive focused on certain targets". The similarity resulted among malware code results due to the fact that a malware is a complex piece of software developed through the use of software engineering principles, the use of program generators, and open-source packages [Walenstein and Lakhotia 2012]. To contribute to cutting edge research, we survey some existing works that belong to binary function identification.

7.1. Binary Searchable Engine

Creating a search engine for executables is an extremely important issue, as it helps reverse engineers to detect the functionality of the code. TRACY [David and Yahav 2014] provides an engine for searching binary functions in the code base. CFGs are decomposed into fixed length subtraces called `tracelets`, and then LCS (longest common subsequence) algorithm is used to align two tracelets. However, the whole structure of the CFG is not taken into consideration. SARVAM [Nataraj et al. 2013] search engine is designed for malware binaries. Given a malware query, a fingerprint is first computed based on transformed image features, and similar malware items from the database are then returned using image-matching metrics. Another search engine called RENDEZVOUS [Khoo et al. 2013], enables indexing and searching binary code using a statistical model comprising `mnemonics`, `control flow sub-graphs`, and `data constant` features. These features are simple to be extracted from a disassembly, yet can be affected by different compilers and optimizations.

7.2. Binary Function Identification

One of the most well-known graph-based approaches, BINDIFF [Flake 2004], treats binaries as directed graphs and puts emphasis on structural similarity rather than the sequence of individual instructions. Building upon BINDIFF, BINSLAYER [Bourquin et al. 2013] uses an approximate bipartite graph matching technique, which benefits from a cost matrix and a Hungarian algorithm that attempts to minimize the edit distance. However, these approaches are not designed to identify FOSS functions in large scale datasets.

BINHUNT [Gao et al. 2008] uses symbolic execution and theorem proving to compare two basic blocks as well as graph isomorphism and backtracking algorithm to find the maximum common included subgraph isomorphism between the CFGs. However, BINHUNT is less practical for analyzing large number of files. Cohen et al. [Cohen and Havrilla 2009] propose a system for discovering similar code. We refer to their system as FH-MCA (Function Hashing for Malicious Code Analysis). Their system is based on using high-confidence hashing algorithms to discover duplicated code in the CERT Artifact Catalog. It also clusters binary programs that have similar regions. However, we believe that applying FH-MCA to FOSS functions identification might have several limitations: First, in our experiments, we observed that the information in the Portable Executable (PE) header (code, data, imports, and other portions of a Windows executable file) might not help in distinguishing FOSS functions, but useful for clustering malware families. Second, we discovered that the control flow graph plays a significant role in identifying FOSS functions; however, this feature is not provided by FH-MCA. Third, malware developers/writers generally modify FOSS packages before they conceal them into their code and this will change the original code, possibly, resulting resulting in different hash signatures. Finally, compiler effects (e.g., optimization settings) and code transportation (e.g., instruction reordering) affect instruction layout by inlining functions, possibly changing register names and modifying instruction sequences by dead code insertion or instruction replacements. Because this will change the binary code of functions, it makes them less detectable by FH-MCA. Another approach called BINHASH [Jin et al. 2012] is proposed. This approach uses semantic hashes for similarity purposes. Specifically, each function is represented by a set of features that are defined as the input-output of a basic block. Then, Min-Hashing is applied for efficient detection. While BINHASH is an effective tool, it has limitations of its own. Specifically, it cannot match code fragments that differ only in the choice of registers [Lakhotia et al. 2013]. Also, the semantics may not be captured completely since not all possible values are used as input [Pewny et al. 2015]. In addition, the CFG is not used, but instead the hashes must be aggregated at the function level. FOSSIL differs from BINHASH in that it extracts the sematic control flow graph walks by applying HSP, which is much less computationally intensive than the process of fuzzing basic blocks used by BINHASH. Yet another limitation of BINHASH is that, as shown in [Pewny et al. 2015], it has a scalability issue when the basic blocks have an input dimension greater than 50. Also, the matching algorithm does not scale well because it attempts to match values such as register names to logic variables by brute force [LeDoux et al. 2013].

A different approach, called BINJUICE, was introduced by [Lakhotia et al. 2013]. It extracts an abstraction of binary block semantics, which is called ”juice”, a term describing the effect of the blocks on the program state. Simple structural comparisons of the function juice are applied to identify semantically equivalent code fragments. However, the authors did not quantify the results, which makes it difficult to evaluate the practicability of their features for FOSS functions identification. A more fundamental problem is that it works only at the basic block level. A new system has been proposed for identifying shared components in a large corpus of malware [Ruttenberg et al. 2014], where a component is defined as a collection of code, to implement a unit of functionality. Although the previous methods that cluster whole programs could be used to cluster FOSS functions, they cannot directly be used for the purpose of identifying FOSS functions, since their focus is on finding components similar to a given component of interest. SIGMA [Alrabaee et al. 2015] has been proposed for identifying reused code in binaries. This technique uses a graph-based representation of code, abstracting away much of the instruction-level detail in favor of structural properties of the program by combining control flow graphs, register flow graphs, and function call graphs. However, this approach does not scale.

7.3. Library Function Identification

Identifying library functions is very important for reverse engineers. Various approaches have been proposed to identify library functions. The well-known library identification technique, IDA FLIRT [Guilfanov 1997], builds the signatures from first 32 bytes of a function with wildcards for bytes that vary when the library is loaded and then applies pattern-matching algorithms for the detection purpose. However, it suffers from two main limitations: signature collision, and sensitivity to any slight differences in the code originating because of various factors, such as small changes in libraries, different compiler optimization settings, or use of various compiler versions. UNSTRIP [Jacobson et al. 2011] identifies wrapper functions in the GNU C library based on the system call interface, since the major part of a program behavior interacts with the OS. However, this approach merely works for wrapper functions and Linux platform; additionally, a library function may have no system call. A novel approach to determine library functions called LIBV is proposed by [Qiu et al. 2016]. First, the execution dependence graphs (EDGs), to introduce the behavioral characteristics of a binary code, is generated. Then, by applying a graph isomorphism and finding similar EDG subgraphs in target functions, both full and inline library functions are identified. However, if various library functions have the same EDGs, LIBV cannot distinguish them. Moreover, compiler optimization settings might eliminate instructions of inlined functions, which may lead to misidentification. Recently, BINSHAPE [Shirani et al. 2017] extracts multi-dimensional features from all library functions, and further by applying feature ranking and decision tree techniques creates best features for each library function (by considering shape of the functions). BINSHAPE detects library functions using modified B^+tree data structure.

7.4. Vulnerability and Bug Identification

Recently, BINGO [Chandramohan et al. 2016] a cross-architecture binary code search is proposed. This system consists of two main components: the first one is designed to filter out OS functions, and the second one generates function models by extracting length variant partial traces. In order to capture the semantics of the functions, library and user-defined functions are inlined into the caller by a selective inlining technique. Most recently, an approach called DISCOVRE has been proposed to do cross-platform bug search [Eschweiler et al. 2016]. This approach extracts various robust features from binary code and then performs searches against the extracted control flow graphs (CFGs). However, the proposed approach is far from being scalable [Feng et al. 2016]. Inspired by DISCOVRE, GENIUS [Feng et al. 2016] 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, and also some changes in the CFG structures affect the accuracy of GENIUS. 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) [Egele et al. 2014], 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 [Eschweiler et al. 2016].

Table XIV: Comparing different existing solutions with FOSSIL

PROPOSALS	Feature				Feature Level				Arch.			Compiler				Ver.
	Syntactic	Semantic	Structural	Statistical	Instruction	Basic Block	Function	Program	x86-64	ARM	MIPS	VS	GCC	ICC	Clang	
FLIRT [Guilfanov 1997]	●	●	●						●	●	●	●	●			
BINDIFF [Flake 2004]		●	●			●	●		●			●	●			●
BINHUNT [Gao et al. 2008]		●	●		●		●		●			●	●			
FH-MCA [Cohen and Havrilla 2009]	●	●					●	●	●			-	-	-	-	
UNSTRIP [Jacobson et al. 2011]		●			●				●				●			
BINHASH [Jin et al. 2012]	●	●					●	●	●			-	-	-	-	
BINJUICE [Lakhotia et al. 2013]		●	●						●				●			
BINSLAYER [Bourquin et al. 2013]		●	●			●	●		●			●	●			●
SARVAM [Nataraj et al. 2013]		●						●	●	●	●	●	●	●		
RENDEZVOUS [Khoo et al. 2013]	●	●	●		●		●		●				●			
BINCLONE [Farhadi et al. 2014]	●				●				●			●				
TRACY [David and Yahav 2014]		●	●			●	●		●				●			
SIGMA [Alrabaee et al. 2015]		●	●			●	●		●			●	●			
LIBV [Qiu et al. 2016]		●				●	●		●			●				
BINGO [Chandramohan et al. 2016]		●	●			●	●		●		●	●	●			
DISCOVRE [Eschweiler et al. 2016]			●						●	●	●	●				
GENIUS [Feng et al. 2016]			●	●					●	●	●		●		●	
BINSHAPE [Shirani et al. 2017]	●	●	●	●			●		●			●	●			
FOSSIL [2017]		●	●		●		●	●	●			●	●			●

Note: The symbol (●) indicates that system supports the corresponding feature, otherwise it is empty.
We use following abbreviations: (Arch.) Architecture, and (Ver.) Verification. (-) means the information is not available

8. LIMITATIONS

In this section, we briefly describe the limitations of our system.

Function Inlining: In practice, the compiler may sometimes inline a small function into its caller code as an optimization. This may introduce additional complexity since we must also be able to fingerprint a function with partial code in another program. We do not currently support such a matching scenario. However, this problem can be circumvented by leveraging data flow analysis to our multidimensional fingerprint. We will study how to systematically address this problem in future work.

Multiple Architecture: Our proposed system deals with only one architecture (x86). We chose this architecture because the most prevalent CPU architectures today are Intel-compatible x86 and x64 CPUs for personal computers and server systems. However, in the world of mobile computing, the ARM architecture is the most common. The MIPS is also important in most control systems. We will study how to systematically address the problem of dealing with multiple architectures in future work.

Type Inference: We currently do not consider type inference in our proposed features. We believe leveraging the concrete data types that are evaluated by the code will aid in verifying the matched functions and consequently reduce the number of false positives. For instance, code with the same control-flow and data-flow may exhibit an integer overflow bug if a critical variable is typed `int32_t` but may not be vulnerable if the variable is an `uint32_t` typed one.

Advanced Obfuscation: We assume that the binary code under analysis is unpacked. While this assumption may be reasonable for many general-purpose software, it implies the need for a pre-processing step involving unpacking before applying the method to malware. Second, our tool fails to handle most of the advanced obfuscation techniques such as virtualization and jitting since our system does not deal with bytecode.

Dataset Size: although our current repository already has a decent size, it would need to be further enriched with a massive number of packages. However, one of the biggest challenges we face involves how to automate gathering, compiling, and indexing FOSS packages. Each FOSS may have its unique dependencies, which makes automating the process difficult. Our future research will include extending this system as a search engine for binary queries, and to also test it under a larger number of FOSS packages. Thus, a small fragment of assembly code or an executable could be queried to obtain useful information related to their functionality.

9. CONCLUDING REMARKS

Identifying FOSS functions is of paramount importance in many security applications; it facilitates the tedious and error-prone task of manual malware reverse engineering and enables the use of suitable security tools on binary code. Determining FOSS functions in malware binaries has received limited attention compared to other fields such as clone detection. Our evaluation demonstrates that our proposed system yields highly accurate results.

Acknowledgment

We would like to thank the anonymous reviewers for providing us very precious comments. Also this article was significantly improved by the insightful comments and suggestions from the anonymous reviewers of IEEE Euro S&P 2017 and IEEE S&P 2017. This research is the result of a fruitful collaboration between the Security Research Center (SRC) of Concordia University, Defence Research and Development Canada (DRDC) and Google under a National Defence/NSERC Research Program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsoring organizations

REFERENCES

2011. HexRays: IDA Pro. (2011). https://www.hex-rays.com/products/ida/index.shtml.

2012. Full Analysis of Flame's Command & Control servers. (2012). https://securelist.com/blog/incidents/34216/full-analysis-of-flames-command-control-servers-27/.

2016. Advanced Windows software protection system, developed for software developers who wish to protect their applications against advanced reverse engineering and software cracking. . (2016). http://www.oreans.com/themida.php.

2016. Adventure in Windows debugging and reverse enigineering. (2016). http://www.nynaeve.net/.

2016a. C++ refactoring tools for visual studio. http://www.wholetomato.com/. (2016). Accessed on Feb, 2016.

2016. DARPA-BAA-10-36, Cyber Genome Program. (2016). https://www.fbo.gov/index?s=opportunity.

2016. EXEINFO PE. http://exeinfo.atwebpages.com/. (2016). Accessed on March, 2017.

2016. HexRays: FLAIR. (2016). https://www.hex-rays.com/products/ida/support/download.shtml.

2016. PELock is a software security solution designed for protection of any 32 bit Windows applications . (2016). https://www.pelock.com/.

2016b. Refactoring tool. https://www.devexpress.com/Products/CodeRush/. (2016). Accessed on Feb, 2017.

2016. Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected. (2016). Accessed on Feb, 2016.

2016. script modifies GNU assembly files (.s) to confuse linear sweep disassemblers like objdump. It does not confuse recursive traversal disassemblers like IDA Pro. It is very inefficient, making simple code about 2x slower. . (2016). https://github.com/defuse/gas-obfuscation.

2016. The Lintian Reports. (2016). https://lintian.debian.org.

2016. The Paradyn project. (2016). http://www.paradyn.org/html/dyninst9.0.0-features.html.

2016. The tracelet system. (2016). https://github.com/Yanivmd/TRACY.

2016. The Z table. (2016). http://www.stat.ufl.edu/ athienit/Tables/Ztable.pdf.

2016. Tigress is a diversifying virtualizer/obfuscator for the C language. (2016). http://tigress.cs.arizona.edu/.

Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2015. SIGMA: a semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation* 12 (2015), S61–S71.

B Bencsáth, L Buttyán, and M Félegyházi. 2012a. Pék, G. sKyWIper (aka Flame aka Flamer): A Complex Malware for Targeted Attacks. *CrySyS Lab: Budapest, Hungary* (2012).

Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. 2012b. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet* 4, 4 (2012), 971–1003.

Daniel Bilar. 2007. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics* 1, 2 (2007), 156–168.

Bencsth Boldizsr. 2012. Duqu, Flame, Gauss: Followers of Stuxnet. (2012).

Martial Bourquin, Andy King, and Edward Robbins. 2013. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 4.

Joan Calvet, José M Fernandez, and Jean-Yves Marion. 2012. Aligot: cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 169–182.

Shuang Cang and Derek Partridge. 2004. Feature ranking and best feature subset using mutual information. *Neural Computing & Applications* 13, 3 (2004), 175–184.

Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2014. Control Flow-Based Malware VariantDetection. *Dependable and Secure Computing, IEEE Transactions on* 11, 4 (2014), 307–317.

Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 678–689.

Cory Cohen and Jeffrey S Havrilla. 2009. Function hashing for malicious code analysis. *CERT Research Annual Report* (2009), 26–29.

Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying dormant functionality in malware programs. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 61–76.

Scott A Czepiel. 2002. Maximum likelihood estimation of logistic regression models: theory and implementation. (2002).

Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. 2016. Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware. *Information Forensics and Security, IEEE Transactions on* 11, 2 (2016), 289–302.

Yaniv David and Eran Yahav. 2014. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 349–360.

Jesse Davis and Mark Goadrich. 2006. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 233–240.

José Gaviria de la Puerta, Borja Sanz, Igor Santos, and Pablo García Bringas. 2015. Using Dalvik Opcodes for Malware Detection on Android. In *Hybrid Artificial Intelligent Systems*. Springer, 416–426.

Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering.

Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.

Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. (2016).

Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. 2014. BinClone: Detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on*. IEEE, 78–87.

Mohammad Reza Farhadi, Benjamin CM Fung, Yin Bun Fung, Philippe Charland, Stere Preda, and Mourad Debbabi. 2015. Scalable code clone search for malware analysis. *Digital Investigation* 15 (2015), 46–60.

Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 480–491.

Eric Filiol and Sébastien Josse. 2007. A statistical model for undecidable viral detection. *Journal in Computer Virology* 3, 2 (2007), 65–74.

Halvar Flake. 2004. Structural comparison of executable objects. *DIMVA 2004, July 6-7, Dortmund, Germany* (2004).

Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Pearson Education India.

Carlos Gañán, Orcun Cetin, and Michel van Eeten. 2015. An empirical analysis of zeus c&c lifetime. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 97–108.

Debin Gao, Michael K Reiter, and Dawn Song. 2008. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*. Springer, 238–255.

Thomas Gärtner, Peter Flach, and Stefan Wrobel. 2003. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*. Springer, 129–143.

Ilfak Guilfanov. 1997. Fast library identification and recognition technology. *Liège, Belgium: DataRescue* (1997).

Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. 2011. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*. ACM, 1–8.

Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 309–320.

Jiyong Jang, Maverick Woo, and David Brumley. 2013. Towards Automatic Software Lineage Inference.. In *USENIX Security Symposium*. 81–96.

Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. 2012. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA), 2012 11th International Conference on*, Vol. 1. IEEE, 386–391.

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015a. Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, Brecht Wyseur (Ed.). IEEE, 3–9. DOI:http://dx.doi.org/10.1109/SPRO.2015.10

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015b. Obfuscator-LLVM: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*. IEEE Press, 3–9.

Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM, 46–53.

Wei Ming Khoo. 2013. Decompilation as search. *University of Cambridge, Computer Laboratory, Technical Report* UCAM-CL-TR-844 (2013).

Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: a search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press, 329–338.

Ivo Krka, Yuriy Brun, Daniel Popescu, Joshua Garcia, and Nenad Medvidovic. 2010. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 179–182.

Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. 2005. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*. Springer, 207–226.

Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. 2013. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 5.

Charles LeDoux, Arun Lakhotia, Craig Miles, Vivek Notani, Avi Pfeffer, and Charles River Analytics. 2013. FuncTracker: Discovering Shared Code to Aid Malware Forensics Extended Abstract. In Proc. 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 2013).

JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

Da Lin and Mark Stamp. 2011. Hunting for undetectable metamorphic viruses. *Journal in computer virology* 7, 3 (2011), 201–214.

Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. CERIAS-Purdue University, 5.

Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. 2012. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 349–358.

Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. 2007. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 431–441.

Ryan McDonald and Fernando Pereira. 2005. Identifying gene and protein mentions in text using conditional random fields. *BMC bioinformatics* 6, 1 (2005), 1.

Jason Milletary. 2012. Citadel Trojan Malware Analysis. (2012).

Ned Moran and James Bennett. 2013. *Supply Chain Analysis: From Quartermaster to Sun-shop*. Vol. 11. FireEye Labs.

Lakshmanan Nataraj, Dhilung Kirat, BS Manjunath, and Giovanni Vigna. 2013. Sarvam: Search and retrieval of malware. In *Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD)*.

Hanchuan Peng, Fuhui Long, and Chris Ding. 2005. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 27, 8 (2005), 1226–1238.

Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-architecture bug search in binary executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 709–724.

Jing Qiu, Xiaohong Su, and Peijun Ma. 2016. Using Reduced Execution Flow Graph to Identify Library Functions in Binary Code. *IEEE Transactions on Software Engineering* 1 (2016), 1–15.

Ashkan Rahimian, Philippe Charland, Stere Preda, and Mourad Debbabi. 2012. RESource: a framework for online matching of assembly with open source code. In *International Symposium on Foundations and Practice of Security*. Springer, 211–226.

Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. 2015. BinComp: A stratified approach to compiler provenance Attribution. *Digital Investigation* 14 (2015), S146–S155.

Brian Ruttenberg, Craig Miles, Lee Kellogg, Vivek Notani, Michael Howard, Charles LeDoux, Arun Lakhotia, and Avi Pfeffer. 2014. Identifying shared software components to support malware forensics. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 21–40.

Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. 2009. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 117–128.

Joshua Saxe, Rafael Turner, and Kristina Blokhin. 2014. CrowdSource: Automated inference of high level malware functionality from low-level symbols using a crowd trained machine learning model. In *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on*. IEEE, 68–75.

Marc Shapiro and Susan Horwitz. 1997. The effects of the precision of pointer analysis. In *Static Analysis*. Springer, 16–34.

Paria Shirani, Lingyu Wang, and Mourad Debbabi. 2017. BinShape: Scalable and Robust Binary Library Function Identification Using Function Shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 301–324.

Mark Stamp. 2004. A revealing introduction to hidden Markov models. *Department of Computer Science San Jose State University* (2004).

Saša Stojanović, Zaharije Radivojević, and Miloš Cvetanović. 2015. Approach for estimating similarity between procedures in differently compiled binaries. *Information and Software Technology* 58 (2015), 259–271.

Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. 2016. Code relatives: detecting similarly behaving software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 702–714.

Annie H Toderici and Mark Stamp. 2013. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques* 9, 1 (2013), 1–14.

S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. 2010. Graph kernels. *The Journal of Machine Learning Research* 11 (2010), 1201–1242.

Andrew Walenstein and Arun Lakhotia. 2012. A transformation-based model of malware derivation. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 17–25.

Chaitanya Yavvari, Arnur Tokhtabayev, Huzefa Rangwala, and Angelos Stavrou. 2012. Malware characterization using behavioral components. In *Computer Network Security*. Springer, 226–239.

Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. 2010. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 95–104.

Yijia Zhang, Hongfei Lin, Zhihao Yang, and Yanpeng Li. 2011. Neighborhood hash graph kernel for protein–protein interaction extraction. *Journal of biomedical informatics* 44, 6 (2011), 1086–1092.

Yijia Zhang, Hongfei Lin, Zhihao Yang, Jian Wang, and Yanpeng Li. 2012. Hash subgraph pairwise kernel for protein-protein interaction extraction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)* 9, 4 (2012), 1190–1202.