SIGMA: A Semantic Integrated Graph Matching Approach For Identifying Reused Functions In Binary Code

By

Saed Alrabaee, Paria Shirani, Lingyu Wang and Mourad Debbabi

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2015 EU

Dublin, Ireland (Mar 23rd- 26th)

Digital Investigation 12 (2015) S61–S71

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

SIGMA: A **S**emantic **I**ntegrated **G**raph **M**atching **A**pproach for identifying reused functions in binary code☆

Saed Alrabaee*, Paria Shirani, Lingyu Wang, Mourad Debbabi

Computer Security Laboratory, Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada

Keywords:
Function identification
Reverse engineering
Binary program analysis
Malware forensics
Digital forensics

A B S T R A C T

The capability of efficiently recognizing reused functions for binary code is critical to many digital forensics tasks, especially considering the fact that many modern malware typically contain a significant amount of functions borrowed from open source software packages. Such a capability will not only improve the efficiency of reverse engineering, but also reduce the odds of common libraries leading to false correlations between unrelated code bases. In this paper, we propose *SIGMA*, a technique for identifying reused functions in binary code by matching *traces* of a novel representation of binary code, namely, the *Semantic Integrated Graph* (*SIG*). The *SIG* s enhance and merge several existing concepts from classic program analysis, including control flow graph, register flow graph, and function call graph into a joint data structure. Such a comprehensive representation allows us to capture different semantic descriptors of common functionalities in a unified manner as graph traces, which can be extracted from binaries and matched to identify reused functions, actions, or open source software packages. Experimental results show that our approach yields promising results. Furthermore, we demonstrate the effectiveness of our approach through a case study using two malware known to share common functionalities, namely, `Zeus` and `Citadel`.

Introduction

The reverse engineering of binary code is generating significant interest among anti-virus companies, security experts, digital forensics consultants, law-enforcement agencies, national security agencies, etc. The objective of reverse engineering often involves understanding both the control and data-flow structures of the functions in the given binary code. However, this is usually a challenging task, because binary code inherently lacks structure due to the use of jumps and symbolic addresses, highly optimized control flow, varying registers and memory locations based on the processor and compiler, and the possibility of interruptions (Balliu et al., 2014).

To assist reverse engineers in such a challenging task, automated tools for efficiently recognizing reused functions and their open source origins for binary code are highly desirable. This is especially true in the context of malware analysis, since modern malware are known to contain a significant amount of library code derived from either standard compiler libraries or open source software packages. The Flame malware, for instance, contains publicly available code packages, including SQLite and LUA (Bencsáth et al., 2012). Hence, the ability to automatically identify reused functions may greatly enhance the effectiveness and efficiency of reverse engineering in such cases.

Existing techniques for identifying reused functions can be roughly categorized into static and dynamic approaches.

☆ This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) of Concordia University, Defence Research and Development Canada (DRDC) Valcartier and Google under a DND/NSERC Research Partnership Program.

* Corresponding author.
E-mail address: s_alraba@encs.concordia.ca (S. Alrabaee).

http://dx.doi.org/10.1016/j.diin.2015.01.011

In a static approach to function identification, different methods have focused on features at different levels (e.g., syntactical, semantical). For example, one existing technique counts mnemonics (opcode names, e.g., `add` or `mov`) in a sliding window over program text (Myles and Collberg, 2005). Another technique discovers exact and inexact clones in binaries through n-grams with normalization (linear naming of registers and memory locations) to address changes in names across different binaries (Saebjørnsen et al., 2009). Recently, an approach combines n-grams with small non-isomorphic sub-graphs of the control–flow graph to allow for structural matching (Khoo et al., 2013). More recently, another approach introduces tracelet-based code search in executables that attempts to statistically locate similar functions in the code base after translating the assembly instructions into an intermediate language (David and Yahav, 2014). While those techniques are not intended to address malware binaries, the authors in Ruttenberg et al. (2014) identify shared software components to support malware forensics. In contrast to most static approaches that focus on one type of features, our approach combines different sources of information into one unified representation of binary code and thus has the potential of producing more accurate results. As to dynamic approaches, since they typically involve executing the code in order to detect the functionality, such approaches usually suffer from prohibitive runtime or exponential growth of execution paths (Calvet et al., 2012; Gröbert et al., 2011).

In this paper, we propose *SIGMA*, a technique for identifying reused functions in binary code by matching traces of a novel representation of binary code, namely, the semantic integrated graph (*SIG*). The *SIG* s enhance and merge several existing concepts from classic program analysis, including control flow graph, register flow graph, and function call graph, into a joint data structure. Such a comprehensive representation allows us to capture different semantic descriptors of common functionalities in a unified manner as traces of *SIG* graphs. Such *SIG* graph traces can then be extracted from binaries and matched, either exactly or approximately, to identify reused functions, actions, or open source software packages.

In summary, our contributions to the problem of identifying reused functions in binary code are as follows.

- We introduce the novel *SIG* representation of binary code to unify various semantic information, such as control flow, register manipulation, and function call into a joint data structure to facilitate more efficient graph matching.
- We define different types of traces such as normal traces, AND-traces, and OR-traces over *SIG* graphs, which are used for inexact matching. We carry out both exact and inexact matching between different binaries, where an exact matching applies to two *SIG* graphs with the same graph properties (e.g. number of nodes), whereas an inexact matching employs graph edit distance to measure the degree of similarity between two *SIG* graphs of different sizes.
- We evaluate our method by experimenting different variants of sort and encryption functions. Experimental results show that our method achieves similarity score close to an optimal similarity matching.
- Finally, we demonstrate the effectiveness of our approach through a case study using two known malware, which share common functionalities, namely, `Zeus` and `Citadel`.

The rest of the paper is organized as follows. Section Existing Representations of Binary Code reviews several existing representations of binary code. Section *SIGMA* Approach provides a detailed description of the main methodology. Section Experimental Results evaluates the proposed approach and compares it to existing work. Section Case Study describes our case study. Section Limitations and Future Direction gives limitations and future directions. Section Related Work reviews related work, and Section Conclusion draws conclusions.

Existing representations of binary code

Numerous representations of binary code have been developed for different purposes of program analysis, such as data flow analysis, control flow analysis, call graph analysis, structural flow analysis, register manipulation analysis, and program dependency analysis. While these representations have been designed primarily for analyzing binary code, they can certainly be employed to characterize the code. In particular, we focus on three representations that capture structural information, namely, control flow graph, register flow graph, and function call graph. These representations form the basis of our approach to identifying reused functions in binary code. For the sake of clarity, we introduce a running example to illustrate these representations using the following sample code (bubble sort).

```
void bubble_sort(int arr[], int size) {
       bool not_sorted = true;
       int j=0,tmp;
     while (not_sorted)
         {
          not_sorted = false;
          j++;
           for (int i = 0; i < size - j; i++){
                   if (arr[i] > arr[i + 1]) {
                       tmp = arr[i];
                       arr[i] = arr[i + 1];
                       arr[i + 1] = tmp;
                       not_sorted = true;
                   }//end of if
                print_array(arr,5);
             }//end of for loop
        }//end of while loop
}//end of bubble_sort
```

Control flow graph

Control Flow Graphs (CFGs) have been used for a variety of applications, e.g., to detect variants of known malicious

applications (Cesare et al., 2013). A CFG describes the order in which basic block statements are executed as well as the conditions that need to be met for a particular path of execution. To this end, basic blocks are represented by nodes connected by directed edges to indicate the transfer of control. It is necessary to assign a label `true (t)`, `false (f)`, or ε to each edge. In particular, a normal node has one outgoing edge labeled ε, whereas a predicate node has two outgoing edges corresponding to a true or false evaluation of the predicate. As an example, the CFG for bubble sort is shown in Fig. 1(a). In our context, CFG is a standard code representation in reverse engineering to aid in understanding the structure of binary. However, while CFGs expose the control flow of a given code, they fail to provide other useful information, such as the way registers are manipulated by the code and the interaction between different functions.

Register flow graph

A Register Flow Graph (RFG) is used to capture how registers are manipulated by binary code, which is originally designed for authorship identification of binary code (Alrabaee et al., 2014). RFGs describe the flow and dependencies between registers as an important semantic aspect of the behavior of a program, which might indicate authorship as well as functionality. We briefly review the concept through an example shown in Fig. 1(b). In the RFG, two labels are assigned to edges; β represents the basic block to which the compare instruction belongs (basic block id), and σ is the cost that is assigned based on the flow of the register values (instruction counts). Regardless of the number or complexity degree of functions, the following registers are often accessed: `ebp`, `esp`, `esi`, `edx`, `eax`, and `ecx`. Therefore, the steps involved in constructing an RFG for these registers are as follows.

- Counting the number of compare instructions,
- Checking the registers for each compare instruction,
- Checking the flow of each register from the beginning until the compare is reached,

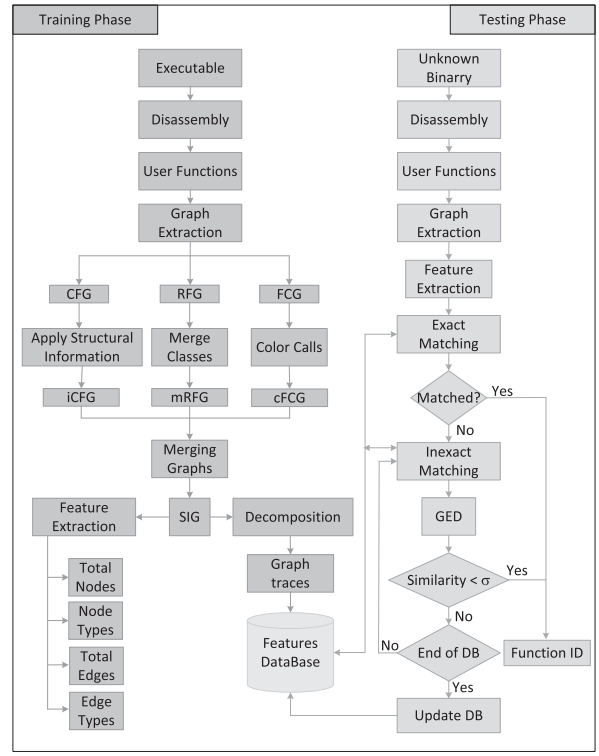

Fig. 2. *SIGMA* architecture.

- Classifying the register changes according to the 15 proposed classes in Alrabaee et al. (2014).

In RFGs, assembly instructions are classified into four families: Stack, Arithmetic, Logical, and Generic operation, as detailed in the following.

- Arithmetic: `add`, `sub`, `mul`, `div`, `imul`, `idiv`, etc.
- Logical: `or`, `and`, `xor`, `test`, `shl`.
- Generic: `mov`, `lea`, `call`, `jmp`, `jle`, etc.
- Stack: `push` and `pop`.

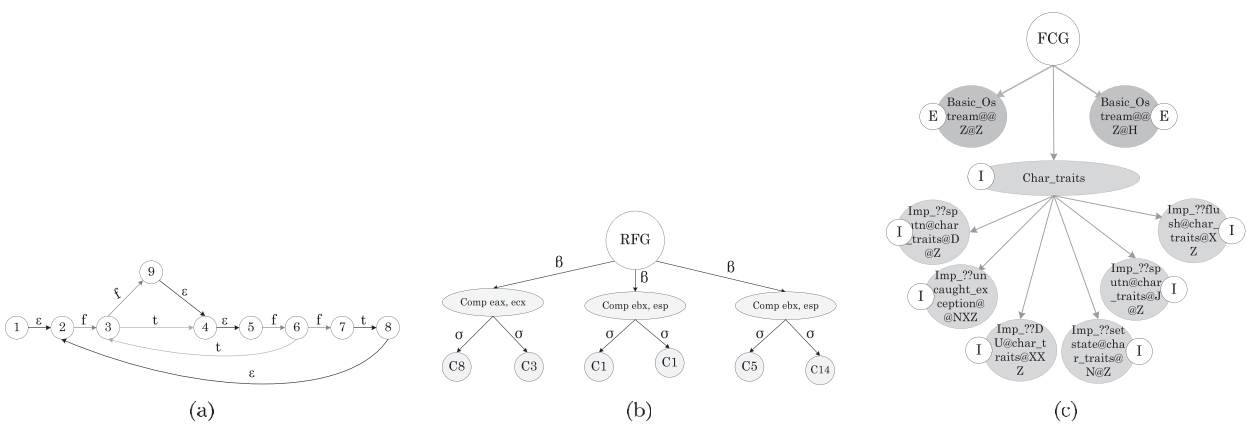

Fig. 1. Classical representations for bubble sort function: (a) Control Flow Graph (b) Register Flow Graph (c) Function Call Graph.

Table 1
Structural information categories.

Category	Description
Data Transfer (DT)	Data transfer instructions such as mov, movzx, movsx
Test(T)	Test instructions such as cmp, test
ArLo	Arithmetic and logical instructions such as add, sub, mul, div, imul, idiv, and, or, xor, sar, shr
CaLe	System call, API call, and Load effective instructions such as lea
Stack	Stack instructions such as push, pop

Function call graph

A Function Call Graph (FCG) is the representation of a function in binary code as a directed graph with labeled vertices, where the vertices correspond to functions and the edges to function calls. Two labels, `I` and `E` are assigned to the nodes; `I` represents internal library functions and `E` represents external library functions. An example of FCG for the bubble function is shown in Fig. 1(c). In the literature, external call graphs have been used for malware detection (Elhadi et al., 2014). In such a case, model graphs and data graphs are compared in order to distinguish call graphs representing benign programs from those based on malware samples (Riesen et al., 2010; Elhadi et al., 2014).

SIGMA approach

In this section, we first provide an overview of the proposed SIGMA approach in Section Overview. We then describe the three building blocks of an *SIG* in Section Building Blocks. We introduce the *SIG* concept in Section *SIG*: Semantic Integrated Graph. Finally, we describe methods for *SIG* graph matching in Section Graph Edit Distance.

Overview

The overall architecture of our SIGMA approach is depicted in Fig. 2. There are two main phases: (i) training phase, and (ii) testing phase, detailed as follows.

The training phase consists of four steps; (i) disassembling the executable and manually filtering out compiler-related functions; (ii) constructing CFG, RFG, and FCG graphs from user functions; (iii) applying structural information to CFG to obtain the informational control flow graph, *iCFG*; applying new merged classes to RFG to obtain a merged register flow graph, *mRFG*; and applying colored classes to FCG to obtain a colored function call graph, *cFCG* (these concepts will be explained in Section Building Blocks). (iv) Merging the previous graphs into a single representation called *SIG*. We then decompose the *SIG* into a set of traces aiming to apply inexact matching between different graphs. Moreover, we consider various properties of the *SIG*, such as the total number of nodes, node types (data, control, dependence, or structural), edge types, total number of edges, the depth of the graph, etc. We save these details into a database with the function ID. On the other hand, given a set of unknown assembly instructions, the testing phase construct the *SIG* and extract the properties of the constructed graph and compare it with the existing *SIG* s graphs in the database. Hence, we have two methods for matching graphs: (i) *exact matching*: two graphs are said to match exactly if they have the same properties. (ii) *inexact matching*: it is based on edit distance calculation and the result is compared to predefined threshold value δ. Two functions are the same if their similarity score is less than δ. More formally, we have following definitions.

Table 2
Color classes for *iCFG*.

Color classes	Majority	Minority
1/2/3	DT, T	ArLo/Stack/CaLe
4/5/6	DT, ArLo	T/CaLe/Stack
7/8/9	DT, CaLe	ArLo/Stack/T
10/11/12	DT, Stack	T/CaLe/ArLo
13/14/15	T, ArLo	DT/CaLe/Stack
16/17/18	T, CaLe	DT/ArLo/Stack
19/20/21	T, Stack	DT/ArLo/CaLe
22/23/24	ArLo, Stack	T/DT/CaLe
25/26/27	ArLo, CaLe	Stack/DT/T
28/29/30	Stack, Cale	T/DT/ArLo

Definition 1. Let f_1, f_2 be two functions, we say f_1 is the copy (or origin) of f_2, if `SIG(`f_1`)` matches `SIG(`f_2`)`.

Definition 2. Let f_1, f_2 be two functions, and `SIG(`f_1`)` $\rightarrow$ `a` and `SIG(`f_2`)` $\rightarrow$ `b` denote extracting *SIG* traces `a` and `b` from f_1 and f_2. Let `sim(a,b)` be a similarity function and δ a predefined threshold value ($\delta < 1$). We say f_1 and f_2 are similar if $sim(a,b) < \delta$.

Building blocks

In this section, we extend the existing representations introduced in Section Existing Representations of Binary Code to form the building blocks of *SIG*.

Structural information control flow graph (iCFG)

As mentioned in Section Existing Representations of Binary Code, traditional CFGs consist of basic blocks each of which is a sequence of instructions terminating with a branch instruction. We can thus only obtain the structure of a function from a CFG. The lack of more detailed information in CFGs means two entirely different functions may yield the same CFG, which will cause confusion for identifying similar functions. Therefore, we extend standard CFGs with a colored scheme based on structural information about the probable role or functionality of each node. For example, if the majority of instructions in one node is arithmetic or logical, it may provide hints about the functionality of the node (e.g., cryptographic function usually involves a large number of `for` loops). By enriching standard CFGs with such information as different colors of nodes, which we call *iCFG*, we have a better chance to distinguish two functions even if they have the same CFG structure. Table 1 shows some example categories of structural information we consider in coloring the nodes.

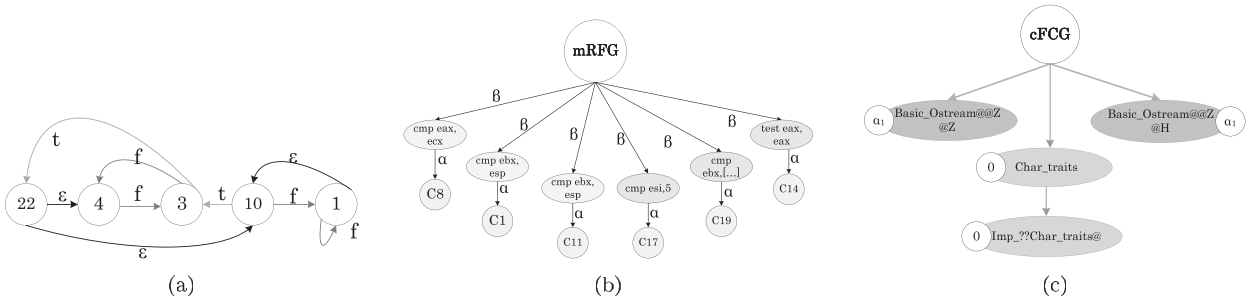

Fig. 3. Enhanced classical representations for bubble sort function: (a) iCFG (b) mRFG (c) cFCG.

Table 3
Updated classes of register access.

Class	Arithmetic	Logical	Generic	Stack	C C	C Reg	ML ML	ML Reg	ML C
1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0
7	1	0	0	1	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0
9	1	1	0	1	0	0	0	0	0
10	1	0	1	0	0	0	0	0	0
11	1	1	1	1	0	0	0	0	0
12	0	1	1	0	0	0	0	0	0
13	0	1	0	1	0	0	0	0	0
14	0	0	1	1	0	0	0	0	0
15	0	1	1	1	0	0	0	0	0
16	0	0	0	0	1	0	0	0	0
17	0	0	0	0	0	1	0	0	0
18	0	0	0	0	0	0	1	0	0
19	0	0	0	0	0	0	0	1	0
20	0	0	0	0	0	0	0	0	1

The assignment of classes depends on two percentages: (i) the two highest percentages, and (ii) the lowest percentage, among the proposed categories. By considering the highest percentages, we aim to measure the majority category in the function. We choose two highest percentages because we have noticed that some classes, such as `Data Transfer`, are always dominant in many cases such that considering in addition the second highest percentage would provide more reliable coloring. Table 2 shows color classes for iCFG. Each row shows three classes. For example, the second row shows classes 1, 2, and 3; class 2 occurs when the two majorities are `DT, T` and the minority is `Stack`.

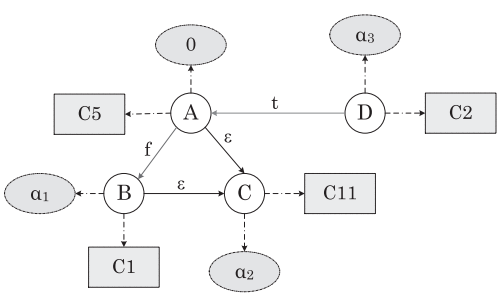

Fig. 4. Simple example of *SIG*.

As an example, by applying the color classes in Table 2 to Fig. 1(a), we can obtain the *iCFG* shown in Fig. 3(a). This *iCFG* involves five color classes: 22, 4, 3, 10, and 1. From Table 2, we can see that the majority of those classes belong to: `ArLo-Stack, DT-ArLo, DT-T, DT-Stack, DT-T`. This is reasonable since the main functionality of the bubble sort algorithm is manipulating values in an array and consequently the main action is transferring the values from one location to another, which explains the large number of `DT` instructions. As demonstrated by the example, by using this extended control flow graph *iCFG*, we can capture more semantic information that might be helpful in identifying functions in binary code. Nonetheless, the *iCFG* only contains control information about basic blocks, and it lacks other useful semantics, such as the way registers are manipulated and the way functions interact with each other. Hence, we introduce two other building blocks in addition to *iCFG*.

Merged register flow graph (mRFG)

As mentioned in Section Existing Representations of Binary Code, *RFG* is a binary code representation for capturing program behaviors based on an important semantics of the code, i.e., how registers are manipulated. The original *RFG* is designed for authorship attribution purposes, therefore it lacks support for some cases that are important for function identification: i) when both operands of `cmp` are constants `(C)`, ii) when one of the operands is a constant and the other one is a register `(reg)`, iii) when both operands are memory locations `(ML)`, iv) when one of the operands is a memory location and the other one is a register, and v) when the operands are a mixture of constants and memory locations. These cases are especially important for identifying functions in binary code, and hence we extend the *RFG* by adding several new classes as shown in Table 3.

Moreover, as another improvement over the original *RFG* representation, we merge certain nodes inside an *RFG*, e.g., class one and class two together are equivalent to class five. In this manner, we can reduce the number of nodes to improve the efficiency in analyzing an *RFG*. Finally, since the original *RFG* depends only on the `cmp` instructions, we also extend RFG instructions to the `test` instruction. After applying those extensions and modifications, we obtain the new representation *mRFG*, as shown in Fig. 3(b). The *mRFG* has three more nodes than its corresponding *RFG*; one of these is `test` instruction, and the other two are related to the immediate memory address and the constant. Moreover, we have merged the original classes (nodes in blue): $C8 - C3$ to $C8$, $C1 - C1$ to $C1$, and $C5 - C14$ to $C11$. The reference to the new classes $C17$ and $C19$ may provide useful semantics about the functions, for instance bubble sort function mainly deals with constants and sorts them in memory locations.

Color function call graph (cFCG)

As mentioned in Section Existing Representations of Binary Code, traditional FCGs represent system calls in a binary code. Among a set of system calls $C = \{C_1, C_2, \ldots, C_n\}$, each call may be either internal or external. To distinguish these from each other, we extend FCGs with a color scheme as follows. The function of coloring nodes defines the color class α in two cases. For an internal call, we only need one color, because internal system calls are mostly related to compiler functions rather than to user functions. As to external calls, we define the color classes using a range of values $0 < \alpha < 1$, because we may have various external system calls potentially connecting to API that is very important for identifying functions. More precisely, we extend FCGs to a new representation which we call *cFCG*, using the color function defined as follows.

$$f(c) = \begin{cases} \alpha = 0, & \text{if } c \text{ is internal system call} \\ 0 < \alpha < 1, & \text{if } c \text{ is external system call} \end{cases}$$

As an example, having applied this new representation to our running example, we obtain the *cFCG* shown in Fig. 3(c). Besides serving as a building block of our proposed approach, the *cFCG* representation may also be helpful in other related tasks by highlighting the difference between various types of calls. This kind of graph may use for malware analysis, for instance, malware clustering through clustering external system calls.

Table 4
Some traces of bubble sort function *SIG*

Node	Traces	Traces type
`22`	ε, ε, C8	out
	t	in
`4`	f, C14	out
	ε, f	in
`22 OR 4`	ε, C8, C14, f	out
	f, t, ε	in
`4 AND 3`	f	out
	f	in

Table 5
Graph Features for bubble sort function.

Features	Frequency
Total # of Nodes	`15`
Total # of Edges	`18`
# of Control Nodes	`5`
# of Control Edges	`8`
# of Call Nodes	`4`
# of Register Nodes	`6`
Connected Graphs	`3`
k-Cone	`1, 2`

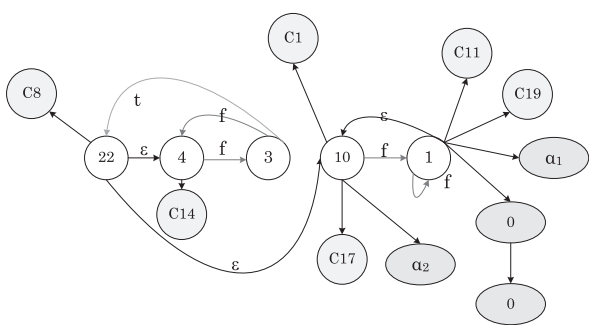

Fig. 5. *SIG* for bubble sort function.

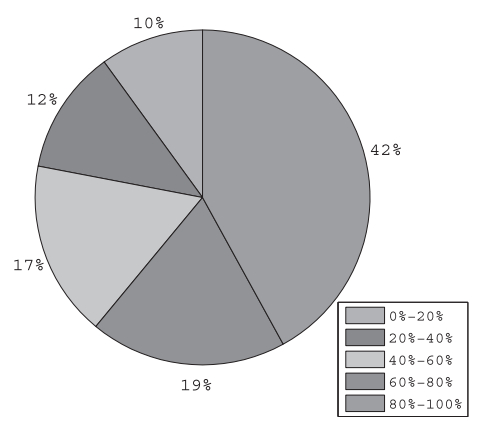

Fig. 6. Similarity scores of function variants.

Table 6
Similarity between sort function variants (B:Bubble sort Q:Quick sort H:Heap sort M:Merge sort).

	B.1	B.2	Q.1	Q.2	M.1	M.2	H.1	H.2
B.1	100%	93%	71%	67%	62%	73%	65%	62%
B.2	96%	100%	79%	80%	70%	72%	60%	68%
Q.1	79%	83%	100%	94%	76%	71%	65%	60%
Q.2	71%	69%	95%	100%	79%	77%	74%	65%
M.1	67%	76%	66%	68%	100%	97%	70%	74%
M.2	73%	69%	77%	78%	94%	100%	70%	72%
H.1	69%	67%	74%	73%	79%	79%	100%	96%
H.2	72%	71%	64%	69%	79%	78%	95%	100%

SIG: semantic integrated graph

The building blocks introduced in the previous section provide complementary views on binary code by emphasizing on different aspects of the underlying function semantics. Inspired by a recent work (Yamaguchi et al., 2014), in which different representations of source code are combined for vulnerability detection in source code (which is a different problem from ours as binary code lack much of the useful information available in source code), we combine those different but complementary representations of binary code into a joint data structure in order to facilitate more efficient graph matching between different binary code for identifying reused functions. Formally, a semantic integrated graph (*SIG*) is defined as follows.

Definition 3. A semantic integrated graph is a directed attributed graph $G = (N,V,\zeta,\gamma,\vartheta,\lambda,\gamma,\psi,\omega,\delta)$ where N is a set of nodes, $V \subseteq (N \times N)$ is a set of edges and γ is a set of labels (e.g. `t`) where ζ is edge labeling function which assigns a label to each edge: $\zeta \rightarrow \gamma$. ϑ is coloring function where each color represents statistical meaning for each node $n \varepsilon N$ based on statistical classes λ. γ is a function for coloring system calls where ψ is a list of external system call labels. Finally, ω is a function for coloring *mRFG* nodes where δ is the list of register classes list.

Fig. 4 illustrates a simple example of *SIG* with four nodes. Note that a *SIG* is a multigraph so two nodes might be connected by multiple edges, e.g., edges corresponding to *mRFG* or *cFCG*. Moreover, `A,B,C,` and `D` represent the outcomes of coloring function ϑ, and `t,f,` and ε are the outcomes of function ζ. Outcomes of function γ are α_1, α_2, and `0`, where `0` represents an internal call, and α_1 and α_2 represent two different external calls. `C1, C5, C2,` and `C11` are outcomes of function ω.

To utilize the *SIG* for inexact matching and matching fragments of a function, we need to consider meaningful subgraphs of *SIG*. Again inspired by Yamaguchi et al. (2014), we decompose a *SIG* into short paths called *traces*, where each trace is represented as ϱ: $S(N) \rightarrow S(N')$ that maps a set of nodes in an *SIG* to another set of nodes according to given criteria, where $S(N)$ denotes the power set of N. The main advantage of such a definition is the composition of multiple traces always yields another trace, i.e., ϱ_0 and ϱ_1 can be chained together to $\varrho_0 \circ \varrho_1$. We define a number of elementary traces that serve as a basis for the construction of other traces, and some examples are shown in the following (each trace function also has other simpler forms, which are omitted due to space limitations).

$$OUT_{M,I,L,K}(Y) = \bigcup_{n \varepsilon Y} \{m : (n,\ m) \varepsilon V,\ \zeta(n,\ m) = M,\ \vartheta(n,\ m) = I,\ \lambda(n,\ m) = L,\ \omega(n,\ m) = K\}$$

$$IN_M(Y) = \bigcup_{n \varepsilon Y} \{m : (n,\ m) \varepsilon V,\ \zeta(n,\ m) = M\}$$

$$OR(\varrho_1, \varrho_2, \ldots, \varrho_n) = \varrho_1 \cup \varrho_2 \cup \ldots \cup \varrho_n$$

$$AND(\varrho_1, \varrho_2, \ldots, \varrho_n) = \varrho_1 \cap \varrho_2 \cap \ldots \cap \varrho_n$$

The trace $Out_{M,I,L,K}$ returns all nodes reachable over edge M and node I. All nodes connected with the node of the other graph with label L and K. Trace In_M represents the in-edge to each node to move backwards in the graph, and the two traces `OR` and `AND` aggregate the outputs of other traces.

Example: SIG for bubble sort function

Here, we give an example of *SIG* for the bubble sort function depicted in Fig. 5. As an example of *SIG* trace, we show the traces of nodes `22, 4, and 3` as well as one example of *OR* and *AND* traces in Table 4. Moreover, we extract additional features as depicted in Table 5. The features in Table 5 include total number of nodes, number of control edges (e.g., `22`), number of call nodes (e.g., `0`), number of register nodes (e.g., `C8`), and etc. Those features together with the *SIG* traces are sufficient for exact matching of *SIG* s, and we will discuss inexact matching in next section.

Graph edit distance

For inexact matching between *SIG* s, we need a distance metric. In this paper, we employ the graph edit distance for this purpose. The edit distance between two graphs measures their similarity in terms of the number of edits needed to transform one into the other (Hu et al., 2009). We implement this concept as follows. Given two *SIG* s, we define the following two elementary traces to transform one graph into another: `Edge-edit traces`, including $\varrho_{\kappa r}$, re-labels the edge, and `Node-edit traces`, including $\varrho_{\nu r}$, re-colors the node by merging nodes from the other graph into one node. An edit edge $V_{G,H}$ between two *SIG* s G and H, is defined as a set of sequence of traces (ϱ_1, ϱ_2, …, ϱ_n) such that $G = \varrho_n\ (\varrho_1(\varrho_{n-1}(\mathrm{H})\ (\ldots\ \varrho_1(\mathrm{H})\ \ldots)))$. To quantify this

Table 7
Similarity between encryption function variants (R:RC4 T:TEA A:AES M:MD5).

	R.1	R.2	T.1	T.2	M.1	M.2	A.1	A.2
R.1	100%	86%	68%	57%	52%	61%	57%	62%
R.2	89%	100%	74%	66%	53%	72%	50%	59%
T.1	72%	79%	100%	87%	66%	61%	55%	67%
T.2	68%	62%	89%	100%	72%	67%	69%	55%
M.1	57%	69%	58%	51%	100%	91%	78%	74%
M.2	63%	67%	67%	70%	92%	100%	78%	72%
A.1	69%	57%	64%	68%	79%	75%	100%	94%
A.2	62%	71%	69%	64%	70%	73%	89%	100%

Table 8
Dissimilarity between sort and encryption functions.

	Bubble.1	Quick.1	Merge.1	Heap.1
RC4.1	86%	93%	79%	87%
TEA.1	96%	91%	79%	89%
MD5.1	79%	88%	90%	94%
AES.1	89%	91%	95%	84%

similarity, the weight of all edit traces is measured, i.e., $V = (\varrho_n, \varrho_2, \ldots, \varrho_n)$ as $w(V) = \sum_{i=1}^{n} w(\varrho_i)$. The edit distance between two *SIG* s is thus defined as the minimum weight of all edit edges and nodes between them, i.e., $sim(G,H) = min\ w(V_{G,H})$. The distance measure between the nodes follows the same reasoning, with operations instead of traces. In Algorithm 1, we calculate the graph edit distance between two *SIG* s G and H, by measuring the cost of transforming G to H. The algorithm starts by extracting the

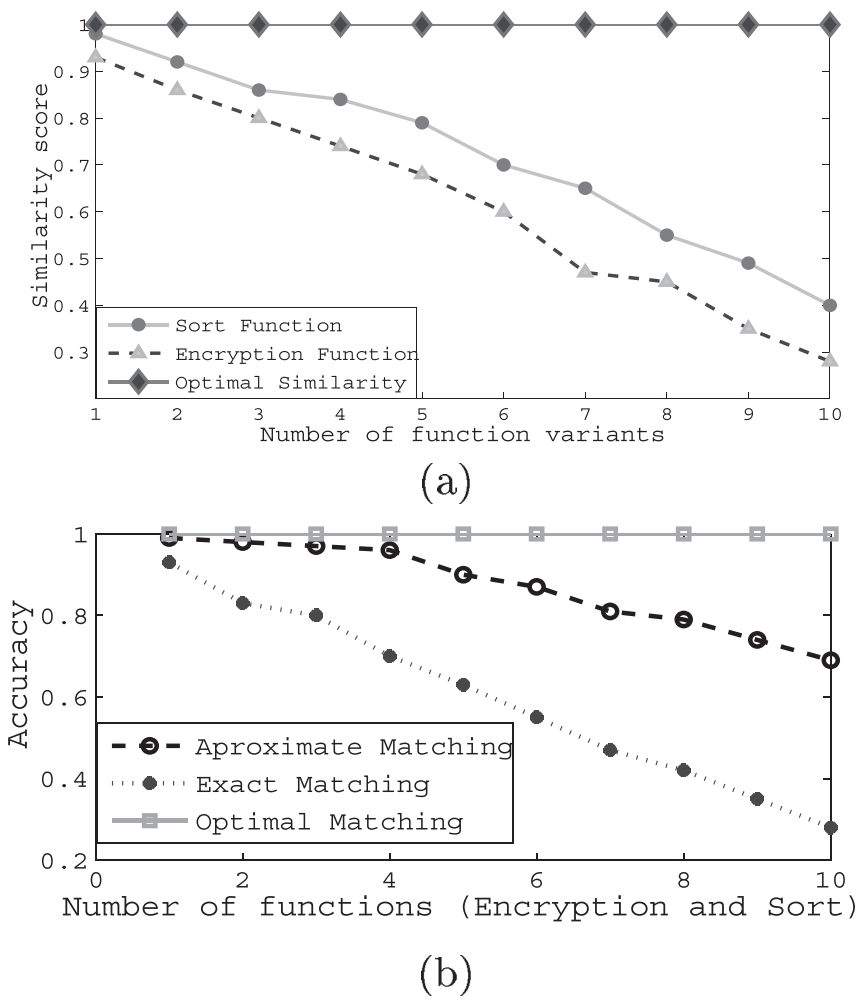

Fig. 7. (a) The relation between the number of variants and the similarity score (b) The accuracy of using exact and approximate matching.

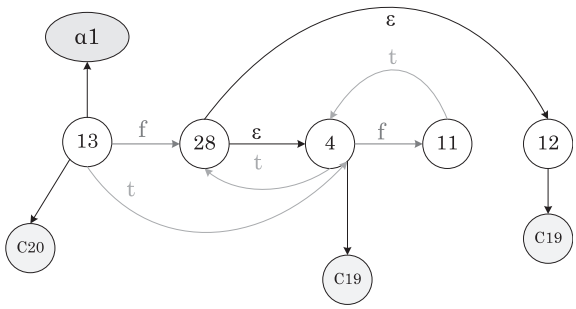

Fig. 8. *SIG* for RC4 in Citadel.

traces of the two graphs as mentioned earlier, and then checks the cost of transforming each node in *G* to nodes in *H*, and finally calculates the total cost.

Algorithm 1: Graph Edit Distance

Input: G, H: Semantic Integrated Graphs (SIG)
Output: sim(G,H): Similarity result between two SIGs
Initialization;
$\varrho_g, \varrho_h \leftarrow \emptyset$; // traces
$w(V_{G,H}) \leftarrow 0$; // weighted cost
$sim(G,H) \leftarrow 0$; // similarity score
begin
  $\varrho_g \leftarrow extract\_traces(G)$;
  $\varrho_h \leftarrow extract\_traces(H)$;
  foreach ϱ_i *in* ϱ_g **do**
    foreach ϱ_j *in* ϱ_h **do**
      $w_j(V_{G,H}) = cost\_of\_transforming(\varrho_i, \varrho_j)$;
    $sim_{i,j}(G,H) = minw(V_{G,H})$;
    $sim(G,H) = sim(G,H) + sim_{i,j}(G,H)$;
  return $sim(G,H)$/i;

We define the dissimilarity between two *SIG* s *G* and *H* as follows.

Definition 5. The dissimilarity $\rho(G,H)$ between two *SIG* s is a value in [0, 1], where `0` indicates the graphs are the most similar and `1` indicates the least similar, as formulated in the following.

$$\rho(G,\ H) = \frac{w(V_{G,H})}{|N_G| + |N_H| + |V_G| + |V_H| + |\varrho_G| + |\varrho_H|}$$

where $w(V_{G,H})$ is the weighted cost of traces, $|N_G|$ and $|N_H|$ are the number of nodes, $|V_G|$ and $|V_H|$ are the number of edges, and $|\varrho_G|$ and $|\varrho_H|$ are the number of traces in *G* and *H* respectively.

Experimental results

We implement and test the proposed technique, *SIGMA*, with variants of sort algorithms and encryption algorithms in order to evaluate the effectiveness and correctness of the proposed method. We employed two variants for each sort algorithm(e.g., bubble, quick, merge, and heap) and each encryption algorithm (e.g., RC4, MD5, Advanced Encryption Standard (AES), and Tiny Encryption Algorithm (TEA)). Using the proposed method, similarity scores amongst these samples are calculated based on the graph edit distance and dissimilarity formulas introduced in previous section. The results are depicted in Fig. 6. A promising value with about 80% similarity score pairs ranging from 0.42 to 1 can be seen in the results. Furthermore, the similarity score on pairs ranging from 0 to 0.2 is only about 10%. The results clearly show that our approach can capture common characteristics between functions relatively well. The occurrences of low-score pairs are mainly due to the significant differences in the sizes of functions and variants, and also the number of nodes, edges, and traces may be observably different. For instance, the number of nodes in a bubble sort variant *a* is 15, whereas for variant *b* is 22; the number of edges in each one is 18 and 43, and the number of traces is 147 and 278, respectively. Table 6 and Table 7, show similarity scores of each pair of sort functions and encryption functions, respectively. The values (100%) in the main diagonal are the similarity scores for the variants when compared to themselves.

We can see from Tables 6 and 7 that similarity scores amongst the sort functions are higher than those among encryption functions. This is due to the fact that the steps of sorting are similar among different algorithms but the steps of encryption functions vary significantly with each algorithm.

In Table 6, the similarities between heap and other algorithms are lower, because the steps of heap sort are significantly different from the other sort algorithms steps. In Table 7, the similarity scores show that RC4 is more similar to TEA, than MD5 is to AES. This is due to the fact that RC4 and TEA have steps in common in the encryption process. In Table 8, the dissimilarity scores between sort algorithms and encryption algorithms are listed. In addition, Fig. 7 shows the relation between similarity score and function variants. It is clear that with more than seven variants, the similarity score is below 0.5, which means that receiving false positive results is potential. Moreover, we can see that our approach yields better results for sort functions than for encryption functions. This is mainly because the steps involved in sorting do not vary significantly with the algorithm used, but those involved in encryption will vary for each algorithm. These may also be different when we have different variants of the same algorithm.

Case study

To demonstrate the effectiveness of our approach when applied to real world binary code, we briefly describe a case study about two well known malware, namely, `Citadel` and `Zeus`. In particular, we would like to identify the stream cipher RC4 function used in both malware. The *SIGs* corresponding to the RC4 function in `Citadel` and `Zeus` are illustrated in Figs. 8 and 9, respectively. The RC4 function in `Zeus` is known to be a reused function from `Citadel` with slight modifications. As can be seen in the figures, the two *SIG* s have common nodes (e.g. 28, 4, and 11). These common nodes were colored based on the majority and the minority of the instruction types, which indicate that these common nodes lead to common actions. In addition, we can observe that both graphs have two common register classes (e.g. C19).

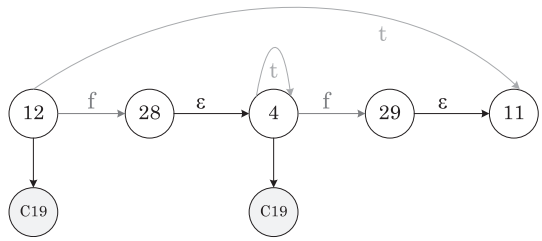

Fig. 9. *SIG* for RC4 in Zeus.

Table 9
Exact matching between *SIG*-RC4 in Citadel and Zeus.

Features	*SIG*-RC4 (Zeus)	*SIG*-RC4 (Citadel)	Similarity
Total # of Nodes	7	9	78%
Total # of Edges	8	11	72%
# of Control Nodes	5	5	100%
# of Control Edges	6	7	86%
# of Call Nodes	1	0	50%
# of Register Nodes	2	3	67%
Connected Graphs	3	3	100%
K-Cone	1,2,3,4	1,2,3	75%
Average Similarity			**78.5%**

Table 10
Inexact matching between *SGF*-RC4 in Citadel and Zeus.

Citadel node	Zeus node	Costs	Node(s) with minimum cost	Cost %
13	12	1 out, 0 in	12	(1/10)
	28	1 out, 1 in	–	
	4	1 out, 2 in	–	
	29	1 out, 1 in	–	
	11	0 out, 2 in	–	
28	12	3 out, 0 in	–	
	28	0 out, 0 in	28 (Select this)	0
	4	3 out, 1 in	–	
	29	0 out, 0 in	29	
	11	0 out, 1 in	–	
4	12	0 out, 0 in	12 (Already chosen)	
	28	1 out, 1 in	–	
	4	0 out, 0 in	4 (Select this)	0
	29	1 out, 1 in	–	
	11	0 out, 0 in	11	
11	12	2 out, 0 in	–	
	28	1 out, 0 in	28 (Already chosen)	
	4	2 out, 2 in	–	
	29	1 out, 0 in	29 (Select this)	(1/10)
	11	0 out, 2 in	–	
12	12	2 out, 0 in	–	
	28	1 out, 1 in	–	
	4	2 out, 0 in	–	
	29	1 out, 1 in	–	
	11	0 out, 1 in	11	(1/9)
Total Cost				**0.311**

More precisely, Fig. 8 illustrates two nodes of *mRFG* within the first four connected nodes which are the same as shown in Fig. 9, which captures the similarity in terms of traces. Based on the *SIG* s and their traces, we conduct exact matching and inexact matching, whose results are shown in Table 9 and Table 10, respectively. The exact matching shows that the similarity is 78.5%, and the cost for inexact matching is 0.311 (close to 0).

Limitations and future direction

The previous section shows that our proposed *SIGMA* approach yields promising results for identifying reused functions in binary code. Nonetheless, the approach still has following limitations which we would like to address in our future work.

- Like most existing approaches, *SIGMA* assumes that binary code is already de-obfuscated. We note that, however, *SIGMA* can in fact address certain forms of obfuscation, such as register reassignments and code recording. Our future work will investigate this potential direction.
- As a learning-based approach, *SIGMA* also requires training data of known functionalities with multiple variants in order to collect sufficient features prior to its application to given code. To this end, we intend to build a feature database for common functionalities by applying *SIGMA* to publicly available code.
- We have not investigated the impact of different compilers in this study. However, we believe that *SIGMA* can overcome some of the changes caused by compilers with the rich set of structural information it captures, and we will confirm this in future work.
- Our future work will also evaluate the capability of the proposed method for dealing with fragments of functions.

Related work

Our main inspiration comes from the recent work of combining different source code representations for discovering vulnerabilities (Yamaguchi et al., 2014). In addition to the idea of combining different sources of information, we also borrow the definitions of graph traces. However, we note that the authors deal with a very different problem, which is to model and detect vulnerabilities, than ours, which is to identify reused functions. Another major difference is that we work on binary code instead of source code. This implies that we must employ entirely different representations, since much of the useful information available in source code, such as abstract syntax trees, is not applicable to binary code. To the best of our knowledge, this is the first effort on the use of multiple sources of structural information for binary code analysis. For identifying reused functions in binary code, existing work mainly fall into two categories: (i) static, and (ii) dynamic. We briefly review static approaches since our work is based on static analysis. Within static analysis, there exist some work that employ graphical representations. Interprocedural control flow using the call graphs of a program have been compared to show similarity to existing malware (Hu et al., 2009), where common nodes between two program call graphs are discovered. The authors in Lee et al. (2010) build their graph by transforming a portable executable into a call graph with nodes and edges that capture system calls and system call sequences, respectively. They then convert the graph to a code graph to expedite analysis. The authors in Elhadi et al. (2014) propose a method where each malware sample is represented

as an API call graph by integrating API calls and operating system resources to represent graph nodes. The authors in Stojanovic et al. (2014) compare metric values and introduce transformers and formulas that could use training data to generate a measure of the similarities between two procedures in binary code. The authors in Xu et al. (2013) propose a method to identify malware variants based on a function–call graph. The authors in (Edler von Koch et al., 2014) develop a pragmatic effective code size reduction technique that exploits the structural similarity of functions. Unlike most existing work, our approach employs multiple sources of structural information to define the distance between variants of functions, which allows for more reliable and efficient matching. Our approach also differs from many existing work in the capabilities of inexact matching and matching function fragments based on graph traces.

Conclusion

The reverse engineering of binary code is an important but challenging task that demands automated techniques for preprosessing and cleaning the code. The identification of reused functions in binary code is one of the important aspect of this issue that has received limited attention in comparison with other aspects of binary analysis. In this paper, we have presented a novel approach called *SIGMA* for effectively identifying reused functions in binary code. Instead of relying on one source of information, our approach combines multiple representations into one joint data structure *SIG*. *SIGMA* also supports inexact matching and exact matching based on traces of the *SIG* which deals with function fragments. Both experimental results and case study have demonstrated the effectiveness of our method, and we have described several potential improvements to the approach in the previous section.

References

Alrabaee S, Saleem N, Preda S, Wang L, Debbabi M. OBA2: an Onion approach to binary code authorship attribution. Digit Investig 2014; 11:S94–103. Elseiver.

Balliu M, Dam M, Guanciale R. Automating information flow analysis of low level code. In: Proceedings of the 21st ACM Conference on Computer and Communication Security, (CCS'14). ACM; 2014.

Bencsáth B, Buttyán L, Félegyházi M, Pék G. sKyWIper (aka Flame aka Flamer): a complex malware for targeted attacks. 2012.

Calvet J, Fernandez JM, Marion JY. Aligot: cryptographic function identification in obfuscated binary programs. In: Proceedings of the 2012 ACM conference on Computer and communications security. ACM; 2012. p. 169–82.

Cesare S, Xiang Y, Zhou W. Malwise–an effective and efficient classification system for packed and polymorphic malware. IEEE Transcation Comput 2013;62:1193–206. IEEE.

David Y, Yahav E. Tracelet-based code search in executables. In: Proceedings of the 35th ACM SIGPLAN conference on Programming language Design and Implementation. ACM; 2014. p. 349–60.

Elhadi AAE, Maarof MA, Barry BI, Hamza H. Enhancing the detection of metamorphic malware using call graphs. Comput Secur J 2014;46: 62–78. Elseiver.

Edler von Koch TJ, Franke B, Bhandarkar P, Dasgupta A. Exploiting function similarity for code size reduction. In: Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems. ACM; 2014. p. 85–94.

Gröbert F, Willems C, Holz T. Automated identification of cryptographic primitives in binary programs. Recent advances in intrusion detection. Springer Berlin Heidelberg; 2011. p. 41–60.

Hu X, Chiueh TC, Shin KG. Large-scale malware indexing using function-call graphs. In: Proceedings of the 16th ACM conference on Computer and communications security. ACM; 2009. p. 611–20.

Khoo WM, Mycroft A, Anderson R. Rendezvous: a search engine for binary code. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press; 2013. p. 329–38.

Lee J, Jeong K, Lee H. Detecting metamorphic malwares using code graphs. In: Proceedings of the 2010 ACM symposium on applied computing. ACM; 2010. p. 1970–7.

Myles, G., Collberg, C., K-gram based software birthmarks, In Proceedings of the 2005 ACM symposium on Applied computing, SAC '05, PP. 314–318, ACM.

Riesen K, Jiang X, Bunke H. Exact and inexact graphmatching: methodology and applications. Managing and mining graph data Advances in database systems, 40. Springer; 2010. p. 217–47.

Ruttenberg B, Miles C, Kellogg L, Notani V, Howard M, LeDoux C, et al. Identifying shared software components to support malware forensics. Detection of Intrusions and malware, and vulnerability assessment. Springer International Publishing; 2014. p. 21–40.

Saebjørnsen A, Willcock J, Panas T, Quinlan D, Su Z. Detecting code clones in binary executables. In: Proceedings of the eighteenth international symposium on Software testing and analysis. ACM; 2009. p. 117–28.

Stojanovic S, Radivojevic Z, Cvetanovic M. Approach for Estimating similarity between procedures in differently compiled binaries, information and software Technology. Elseiver; 2014.

Xu M, Wu L, Qi S, Xu J, Zhang H, Ren Y, et al. A similarity metric method of obfuscated malware using function-call graph. J Comput Virology Hacking Tech 2013;9(No 1):35–47. Springer.

Yamaguchi F, Golde N, Arp D, Rieck K. Modeling and discovering vulnerabilities with code property graphs. In: Proc. of 35th IEEE Symposium on Security and Privacy. IEEE; 2014.