Digital Investigation 12 (2015) S61–S71

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2015 Europe

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