ASSEMBLY TO OPEN SOURCE CODE MATCHING FOR REVERSE ENGINEERING AND MALWARE ANALYSIS

ASHKAN RAHIMIAN

A THESIS

IN

THE CONCORDIA INSTITUTE FOR INFORMATION SYSTEMS ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF APPLIED SCIENCE (INFORMATION SYSTEMS SECURITY)

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

SEPTEMBER 2013

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ashkan Rahimian**

Entitled: **Assembly to Open Source Code Matching for Reverse Engineering and Malware Analysis**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Information Systems Security)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. A. Ben Hamza _____ Chair

Dr. Jeremy Clark _____ Examiner

Dr. W. Hamou Lhadj _____ External (to program)

Prof. Mourad Debbabi _____ Supervisor

Approved _____

Prof. Rachida Dssouli, Director

Concordia Institute for Information Systems Engineering

_____ 2013 _____ _____

Prof. Christopher W. Trueman, Dean

Faculty of Engineering and Computer Science

Abstract

Assembly to Open Source Code Matching for

Reverse Engineering and Malware Analysis

Ashkan Rahimian

The process of software reverse engineering and malware analysis often comprise a combination of static and dynamic analyses. The successful outcome of each step is tightly coupled with the functionalities of the tools and skills of the reverse engineer. Even though automated tools are available for dynamic analysis, the static analysis process is a fastidious and time-consuming task as it requires manual work and strong expertise in assembly coding. In order to enhance and accelerate the reverse engineering process, we introduce a new dimension known as clone-based analysis. Recently, binary clone matching has been studied with a focus on detecting assembly (binary) clones. An alternative approach in clone analysis, which is studied in the present research, is concerned with assembly to source code matching. There are two major advantages in considering this extra dimension. The first advantage is to avoid dealing with low-level assembly code in situations where the corresponding high-level code is available. The other advantage is to prevent reverse engineering parts of the software that have been analyzed before. The clone-based analysis can be helpful in significantly reducing the required time and improving the accuracy of static analysis. In this research, we elaborate a framework for assembly to open-source code matching.

Two types of analyses are provided by the framework, namely online and offline. The online analysis process triggers queries to online source code repositories based on extracted features from the functions at the assembly level. The result is the matched set of references to the open-source project files with similar features. Moreover, the offline analysis assigns functionality tags and provides in-depth information regarding the potential functionality of a portion of the assembly file. It reports on function stack frames, prototypes, arguments, variables, return values and low-level system calls. Besides, the offline analysis is based on a built-in dictionary of common user-level and kernel-level API functions that are used by malware to interact with the operating system. These functions are called for performing tasks such as file I/O, network communications, registry modification, and service manipulation. The offline analysis process has been expanded through an incremental learning mechanism which results in an improved detection of crypto-related functions in the disassembly. The other developed extension is a customized local code repository which performs automated source code parsing, feature extraction, and dataset generation for code matching. We apply the framework in several reverse engineering and malware analysis scenarios. Also, we show that the underlying tools and techniques are effective in providing additional insights into the functionality, inner workings, and components of the target binaries.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Prof. Mourad Debbabi for his enlightening guidance, persistent support and encouragement during my studies. His guidance and immense knowledge have helped me further develop my research quality and professional skills. Without his support, this thesis would not have been possible.

Furthermore, I would like to thank the thesis committee: Dr. Jeremy Clark, Dr. W. Hamou Lhadj and Dr. A. Ben Hamza for their insightful, valuable and detailed comments.

This research was conducted in collaboration with DRDC Valcartier. I would like to acknowledge Mr. Philippe Charland and Mr. Martin Salois for their encouragement and practical advice during the design and development of the RE-Source framework. Also, I would like to thank ESET Canada for their collaboration on malware deobfuscation and acknowledge the help and support provided by Mr. Marc-Étienne Léveillé and Mr. Pierre-Marc Bureau.

I would like to acknowledge the help of Dr. Stere Preda at different stages of my research. I am also grateful to the entire Concordia University family. Specifically, I would like to thank Dr. Laurie Lamoureux Scholes, the Director of the GradProSkills Team.

Last but not the least, I am grateful to my parents for their support and unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The present research addresses the problem of assembly to open source code matching, which is an important aspect of reverse engineering and static malware analysis. Generally, software reverse engineering is conducted in situations where the source code of an application is not available. For instance, software audits are performed on executable binaries and release versions to assess the conformance with design and quality specifications. Similarly, malware reverse engineering is another domain where detailed information is required on the behavior and functionality of malware files. Two major reverse engineering approaches are common in this context, known as static and dynamic. The static analysis approach focuses on assembly-level functions for performing tasks such as data and control flow analysis. The disassembly is studied without executing the program. Stack frame information, execution paths, and assembly functions convey important information during the static analysis. On the other hand, the dynamic analysis approach is concerned with the actual execution of the program and observation of its behavior and interaction with the operating

system. When working with untrusted applications, the analysis is often done in a controlled environment such as a virtual machine. A combination of both approaches are required for getting a complete picture of the program code and behavior. During the dynamic analysis, the live values of registers and variables can be monitored and altered in memory.

Reverse engineering is a time-consuming task and it poses many challenges. In case of static analysis, the disassembly consists of a large number of instructions in which data and code are interleaved. The low-level representation of the program may not convey a clear idea of the program behavior due to obfuscation and encryption. Therefore, a great amount of manual work should be performed by the reverse engineer during the analysis. The successful outcome of the analysis is tightly coupled with the functionalities of the tools and skills of the reverse engineer. Due to the enormous number of assembly instructions, going through them, line by line is not often a feasible solution. Therefore, a reliable and scalable approach is needed for providing a high-level picture of the disassembled code. Then, the reverse engineer would be able to focus on interesting low-level details only as required. This approach will reduce the required time for analysis and improve the accuracy and reliability of the results. As an example, the disassembled version of a Win32 binary file with a size of 150KB might result in an assembly file with more than 100,000 instructions.

One way to reduce the manual work is to utilize a clone-based analysis on the disassembly. Since code reuse is common in application development, shared code and libraries can be identified quickly through an automated clone search approach. Thus, the analysis focus is shifted to the non-library code portions, which results in considerable savings in time and efforts spent by the reverse engineer. To enhance and accelerate the malware analysis process, another dimension is considered in our study as shown in Figure 1. This new dimension is called clone-based analysis.

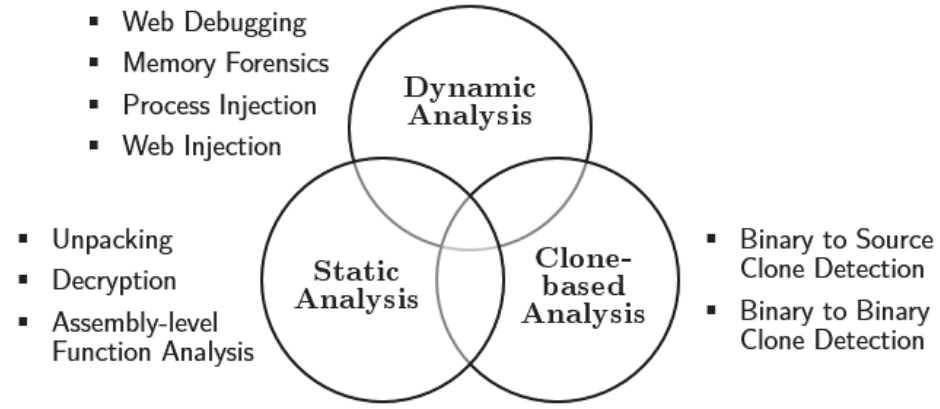


Figure 1: Overlap in Reverse Engineering Methodologies

In few words, the clone-based analysis identifies the pieces of code that are originated from other malware and open-source applications.

The main objective of this research is to elaborate a framework for automated assembly to source code matching. We devise methodologies and propose algorithms that help reverse engineers get faster insights into the internal structure and inner workings of the programs under analysis. Furthermore, we design and implement the necessary tools and techniques for a practical assembly to source code matching platform. Also, we test and evaluate the framework under several real-world reverse engineering scenarios.

1.1 Problem Statement

Clone-based methodologies use similarity metrics for measuring the distance between the clones. These approaches have applications in plagiarism detection, source code clone detection, and assembly clone detection. Even though binary clone matching techniques are used in reverse engineering research, assembly to source code matching techniques are still relatively unexplored.

These techniques can be used for comparing the assembly of one source to the source of anther.

This research gap provides an opportunity for further studies in this area. One major advantage of the assembly to source matching is that it provides insightful hints into the open-source components and standard libraries used in the disassembly. As a result, the reverse engineer would be able to circumvent dealing with low level assembly code when similar high level source code is available. Working with source code is preferred by the reverse engineers due to parameters such as readability and performance.

The other benefit of this approach is that it helps in unleashing the overall code context and potential functionality. This is achieved through analyzing the library and system calls, classification of functions based on API groups, and additional commenting. Decompilation is a common technique in reverse engineering, which aims at retrieving high-level source code from binaries. Since the compilation options and optimization affect the reversibility of the process, the decompiled files are often very different from the original sources. Assembly to source code matching techniques can also be used for complementing and enriching the decompilation results by providing links to similar source files.

1.2 Motivation

Due to the complex structure and the advanced anti-reverse engineering techniques utilized by modern malware, the analysis process is often prolonged. This allows cyber criminals to carry on with their attacks while the analysis is still in progress. Therefore, new tools and techniques are required for accelerating the reverse engineering process and for providing additional insights into

the functionality, inner workings, and components of the malware.

The first part of the present research pertains to the domain of static assembly code analysis, and more precisely in mapping of assembly to source files. Although a decompiler seems to be the program, which best fits our goals, we consider that an attempt of mapping the assembly to existing source code could complement the decompilation and disassembly results. Compilation is usually a lossy and one-way process in which many source level features are discarded due to compilation options and optimization. Applications such as malware analysis can benefit from a tool that is able to provide reliable information about the standard and open source files used by a malicious developer. Such information can help the reverse engineer save a significant amount of analysis time by finding references to available source code and algorithms. It can also provide hints for unleashing the overall context of the assembly code.

In case of cryptographic algorithms or obfuscated code, delving deep into the individual assembly instructions might not yield meaningful results. The disassembled code of such algorithms contains different kinds of mov, xor, shift, and rotate operations such as shr, shl, ror, rol, etc. In contrast, focusing on interesting blocks of assembly instructions provides more hints into revealing the underlying algorithm. Besides, it requires less time spent on the machine-dependent assembly instructions. The winning strategy in reverse engineering is to draw a global view of the code first, and delve into the details only as needed.

Decompilers, methods and tools to analyze malware code already exist [41]. They can be used by expert reverse engineers who seek to understand the origins and the creation process of the malware. *IDA Pro* [59] allows disassembling a binary file and its rich and interactive GUI simplifies the analysis process. It is widely used thanks to its heuristic disassembly, cross-platform

compatibility, the integrated debuggers, and, more interestingly, plugins such as the *Hex-Rays* decompiler - "the most advanced decompiler ever built!" [12]. *IDA Pro* can be used in security, as well as general (non-security) applications.

Although there have been a few attempts to design generic debuggers to work on heterogeneous platforms (e.g., *GenDbg* [4]), *IDA Pro* proves to be one of the most comprehensive tools in reverse engineering. Two algorithms are common in generating assembly files from machine code instructions namely *Linear Sweep* and *Recursive Descent* [44]. As it will be discussed in later chapters, these algorithms are vulnerable to anti-disassembly techniques used by malware. As a results, the generated disassembly might lack certain functions or it might not distinguish between data and code properly. The assembly code follows rather regular patterns and consequently the decompiler is able to do a mapping between registers, memory locations, and abstract variables to C-like statements. Yet, most of the decompilers are not generic. Other basic C constructs (e.g., loops) are more difficult to extract and some decompilers fail to recognize them (e.g., *Boomerang* [2]). Another challenging problem is reconstructing the abstract types (e.g., structures) and *TyDec* [13] tries to tackle the problem but is limited to an experimental level. In this case, the best practice remains the human expertise for manually defining the structures and abstracting the program elements.

Our approach is rather different in that our *RE-Source* framework is meant to inform the reverse engineer about the standard and open source components that might have been used by the creator of the binary file. To our knowledge a similar functionality was only provided by the *RE-Google* IDA Pro plugin developed by Felix Leder [9]. We draw inspiration from this project for developing the *online analysis* module of *RE-Source*. In contrast to the other project, *RE-Source* performs both *online* and *offline analysis* on the disassembly. *RE-Google*, written in Python, relies on the

IDA API and the Google Code Search API [7]. It takes the disassembled binary code as input and creates a query for submitting to Google Code Search based on the constants, strings, and function names. The response from the search engine is the potential source excerpt that contains similar features. Although it supports a limited set of features to create a query, *RE-Google* may confine the search to certain languages. Additionally, it can be configured to search for a specific function within the disassembly, skip certain functions, or perform a search for all available functions. Also, the interval between two subsequent searches can be defined. Optionally, user credentials could be supplied as part of the query to the code search engine. Furthermore, there is an option for restricting certain string patterns in the result. A constant filter function checks the immediate values and removes flags and small values from the query if they are not relevant for the search. The response from the search engine is parsed by the Google API and the top results are added to the code as comments. *RE-Google* was designed on top of the *GData framework* and *Google Code Search* APIs [7], which were officially deprecated in 2010. The GData framework provides a REST-based interface for communicating and exchanging information over the web. Our framework provides a functionality similar to *RE-Google* in online analysis and introduces new ones for offline analysis. Besides, it does not rely on GData as it has built support for query processing and parsing.

1.3 Objectives

The main objective of the thesis is to elaborate a practical framework for assembly to source code matching that will significantly reduce the required time and effort for static analysis and enhance the reverse engineering process. More explicitly, the thesis objectives are:

- Devise an expandable online analysis mechanism for identifying the shared and open source components and standard algorithms based on extracted features from the disassembled code. The analysis process must not rely on direct code search engine APIs such as GData or similar frameworks.

- Design and implement the main components and building blocks of a scalable and language-independent framework for assembly to source code matching.

- Elaborate the framework through an offline analysis mechanism for providing insights into the potential functionality of assembly functions and their interaction with the operating system.

- Apply the methodology and the framework in real-world reverse engineering scenarios.

With the principle of *code reuse* in mind, the framework should exploit some features that exist at both the source and the assembly file levels. Also, it should be able to trigger queries based on these features on certain repositories used by the developers' community. Besides, the framework should take into account a large panel of search engines for the online analysis. Likewise, an offline analysis module must be designed for providing information regarding the functionality of a portion of the assembly file even if the online analysis results were not available. The offline analyzer module should report on function stack frame, prototype, arguments, local variables and low-level system calls. Moreover, it should contain a built-in dictionary of user-level and kernel-level API functions that are commonly used by malware to interact with the Windows operating system for performing tasks such as file I/O, network communications, registry modification, and service manipulation.

1.4 Approach Overview

Open source libraries are being used in many applications. Recently, an upwards trend has been seen in the use of open source cryptographic and network-related libraries in well-known malware [46]. Therefore, it is desirable that such components be identified quickly before the process of reverse engineering is initiated [40]. The *RE-Source* framework is a solution for the online matching of assembly with open source code [47], which takes into account the idea of code reuse and highlights the common open source components in binary files. The system examines each assembly-level function in two phases of online and offline analysis. The key steps of the framework are as follows: (1) Extraction of interesting features, (2) feature-based query encoding, (3) query refinement for online code search engines, (4) request/response processing, (5) data extraction and parsing, (6) reporting results and updating comments, (7) feature-based offline analysis. Different set of features are considered for online and offline analysis. It can be shown that a consistency is observed among the online and offline results of malware analysis when the malicious code makes use of user/kernel-level Windows API functions[1]. As an expansion to the framework, a new module for offline analysis is introduced that improves the accuracy of matching by incremental learning of the mappings between assembly-level and source-level feature vectors. Figure 2 shows an overview of the *RE-Source* framework components. To evaluate the solution, the *RE-Source* framework is used in several real-world reverse engineering scenarios.

The system consists of several modules for interacting with the disassembly and code repositories. The *Src* table stores the extracted source level features and the *Bin* table contains the assembly information. A pattern matching technique scans the index of joint feature vectors for classification

[1]The framework was tested on the Windows platform.

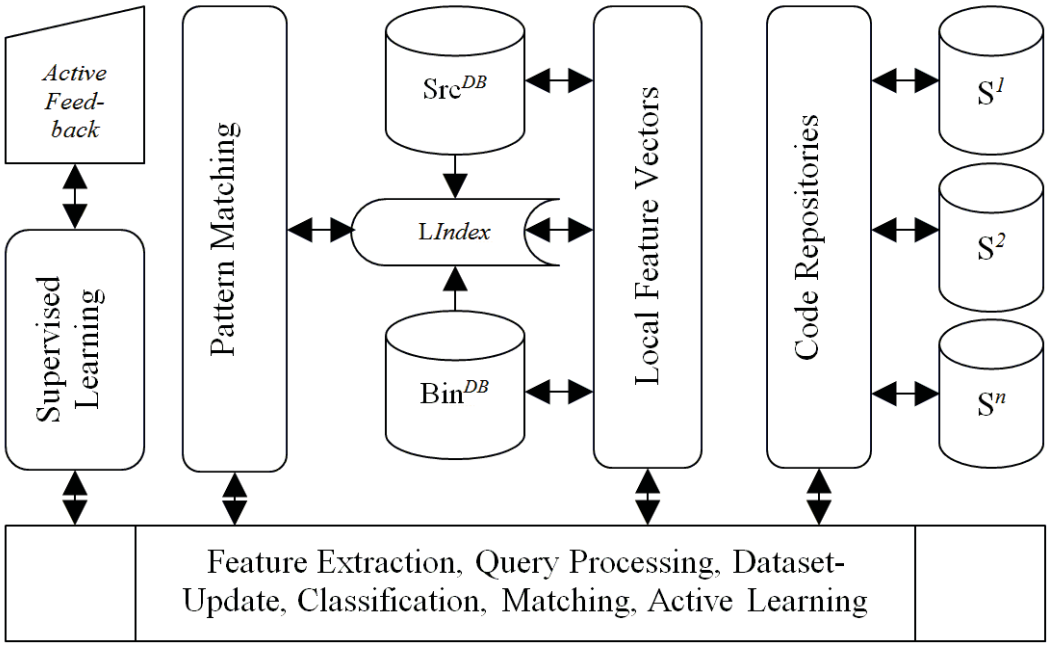


Figure 2: Overview of the RE-Source System Architecture.

and assigning functionality tags. Furthermore, an incremental learning module has been employed as an offline analysis extension to the framework for improving the accuracy of the results through an active learning mechanism.

1.5 Contributions

The major contributions of this thesis can be summarized as follows:

1. Elaboration of the *RE-Source* framework along with the algorithms, components and methodology for online matching of assembly with open source code. This framework has been implemented as a GUI IDA Pro Python plug-in. It can be used for finding references to open source code which shares certain features with the disassembly under analysis. It also

generates a dataset for based on the assembly level features.

2. Definition, design and implementation of an *Offline Analysis Module* for assembly function analysis based on functionality tags for common malware behavior. The tagging module has been integrated with the main GUI and the results are immediately applied to the IDA Pro's function window of the disassembly. Also, the matched references are added as function comments to the disassembly. Moreover, this module provides informative statements about assembly functions to the reverse engineer.

3. Elaboration of the data-mining extension for the offline module through incremental learning and function semantics for malware analysis. Using a machine learning and data-mining Python package, the system is able to learn functionality patterns from the analyzed malware disassemblies. The learning module improves the precision of the function classification process. This module is shown to be effective in detection of crypto-related functions based on constants and the sequence of function calls.

4. Design and development of a customized local code repository for parsing the source code, feature extraction, and automated dataset generation. The local repository stores the source code which are used in the matching process. Besides, the search process can be customized based on specific file extensions. The current prototype support parsing standard C/C++ source files for extracting source level features. The process then generates datasets for the matching process. Each row in the assembly dataset corresponds to a function in the disassembly and contains all the extracted features. Similarly, the source dataset stores the class level information.

5. Leveraging the developed tools and the proposed clone-based methodology in several reverse engineering scenarios such as the infamous Citadel malware. *RE-Source* is able to reveal additional malware functionalities such as video capturing and key-logging by tagging the functions with the relevant labels. Furthermore, it allows us to quantify the similarities between different variants of the malware.

Our first proposal seems to share a common goal with a research trend, which is bottom-up analysis of assembly, i.e., decompiler. Yet, we do not propose a decompiler but a tool to match assembly with existing open source. To our knowledge, only one discontinued solution was similar to our approach, *RE-Google*. We presented our methodology in [47]. In order to enhance the process of matching assembly with source, we go beyond the online analysis and propose an offline one with data-mining techniques. Without demonstrating the applicability of the approach in real world scenarios, the research potential cannot be unveiled. For this, we have applied the methodology on the infamous Citadel malware.

1.6 Organization

This thesis is organized as follows. The background and related work are introduced in Chapter 2. The proposed approach for the elaboration, design and implementation of the *RE-Source* framework is discussed in Chapter 3. The learning algorithms, elaboration details, and platform extensions are presented in Chapter 4. A comprehensive reverse engineering case study is reported in Chapter 5 and finally, the concluding remarks about this research are formulated in Chapter 6.

Summary

This chapter introduced the research area by discussing the problem statement, motivation and objectives. It also presented an overview of the proposed approach and enumerated the main contributions of the thesis. In the next chapter, the background and related work will be presented for setting the study context.

Chapter 2

Background and Related Work

This chapter sets the research context by introducing the related work and highlighting the background. This work can be classified in the overlapping area of reverse engineering, malware analysis, and clone detection. The proposed methodologies in the following chapters assume basic familiarity with these topics. For this reason, the relevant tools and techniques are discussed for assembly to source code matching, recognizing libraries and program elements, measuring the code similarities, pertinent malware analysis and detection techniques, and reverse engineering reports on the Citadel malware.

This chapter is organized as follows. An overview of the malware reverse engineering techniques is given in Section 2.1 followed by the introduction of the related work and the Citadel malware analysis. Section 2.2 presents the binary and source clone-detection approaches and the tools used in deep analysis of binary files.

2.1 Reverse Engineering Malware

Reverse engineering is a term given to the process of discovering the design specifications and implementation details with little or no knowledge about the production procedure of a subject system. It can be done for many reasons such as security auditing, interfacing with other systems, vulnerability analysis and bug fixing. From a software engineering point of view, reverse engineering is similar to black-box testing. In malware reverse engineering context, three types of analysis namely static, dynamic, and memory are performed on a malware binary in order to discover its functionality and behavior. Dissecting the malware is a required step for mitigating malware threats.

2.1.1 Malware Analysis Techniques

Three common types of forensic analysis techniques namely functional, relational, and temporal are utilized in malware investigations [32]. These techniques are applied for gaining a deeper understanding of the potential capabilities and actual behavior of a malware. The aim of functional analysis is to monitor the dynamic behavior of a piece of malware in a controlled analysis environment. The process of dynamic analysis might not always be straightforward. Advanced malware could possess anti-reversing (anti-static, anti-dynamic, and anti-virtual machine) mechanisms for evading static and dynamic analysis and hindering the process [41, 44]. The malicious code can sense whether it is being debugged and analyzed in a virtual environment. The work in [22] studies malware analysis in such scenarios and proposes a transparent mechanism for preventing the

malware from spotting the API hooking. They also present techniques for malware library call detection. The purpose of relational analysis is to study the internal components of the malware and the interactions with the external environment. Furthermore, the temporal information is captured by temporal analysis. A combination of static and dynamic analysis techniques should be used for handling packed malware, code de-obfuscation, and data decryption.

Static analysis is performed for understating the inner-workings of the malware. In this analysis, the binary is transformed into assembly instructions such as x86, x64, and ARM. Each of these platforms have a different set of assembly instructions. Expertise in assembly instructions are essential in static analysis. The x86 architecture can be abstracted into three modules of CPU, I/O, and RAM. Four types of registers are used by the CPU, namely general, segment, flags, and instruction pointers. The flags or status registers are important in malware analysis as they can change the execution path of the program. The program memory is divided into four sections namely stack, heap, code, and data. The stack is where the local variables and function parameters are stored and the heap is used for dynamic memory.

The most important aspect of static analysis is to correctly recognize the high level code constructs. Global and local variables are represented differently in the assembly. The global variables are referenced by memory addresses whereas local variables are placed on the stack. Control structures and conditional statements can be recognized by different forms of jump instructions. Also, grouping the instructions into categories of initialization, comparison, and stepping is helpful in finding the loop blocks. The other important aspects of the analysis are function calling convention, recognition of structs, arrays, and linked lists, finding class constructors, destructors, and special data structures such as virtual function tables.

Dynamic analysis is performed to understand the behavioral aspects of the malware. Low level debuggers such as *OllyDbg*, *Immunity Debugger*, *WinDbg*, and *IDA Pro* are common in practice [41, 44]. Debuggers are used to view and change the values of registers and memory data. Particular instructions can be patched in live memory of the malware in order to change the behavior or explore other malware functionalities. Several breakpoints are set on important assembly instructions and the debugging process is paused when the breakpoints are hit. Two types of software and hardware breakpoints are commonly used during the debug. The software breakpoint is equal to the `INT3` or the `0xCC` instruction. Upon the execution of this instruction, the operating system throws an exception and transfers the control to the debugger. Hardware breakpoints are implemented through dedicated hardware registers and are more robust in dealing with anti-debugging techniques used in malware. Debugging is an essential step for unpacking and deobfuscation. It is also important in generating network signatures for the malware. The detailed process of Citadel dynamic analysis will be presented in Section 5.4.

Memory forensics is another dimension in the malware reverse engineering process. The two steps involved in the process are memory acquisition and memory analysis. An image is captured from the malware process memory for extracting certain artifacts or values such as encryption keys, initialization seeds, and loaded modules. Dynamic memory can be dumped into executable file images and analyzed statically. For instance, the Volatility framework [63] is an effective tool for this purpose. It is able to generate executable versions of the running processes and search in them according to user-specified criterion. Section 5.4.2 elaborates on Citadel memory forensics and provides examples of the data that are obtained from physical memory.

2.1.2 Malware Detection Approaches

The most common malware detection techniques rely on hash-based signatures. This scheme can be easily bypassed by malware through slight modifications in byte level information. Polymorphic malware can generate several variants based on a similar core. Therefore, detection techniques, which depend on constant features of malware are more reliable. One such feature is the sequence and type of API calls in a malware. API monitoring can be done in both the static and dynamic analysis scenarios. By considering the semantics behind the system and API calls, the investigator would be able to gain more insights into the intent and inner workings of malware components. As opposed to signature-based approaches, heuristic-based techniques are shown to be more robust and promising in detecting previously unknown malware samples [39, 40]. The study in [38] utilizes data mining techniques on different opcode sequences that are extracted from both malicious and benign binaries. Also, they perform feature frequency analysis based on weighted terms for selecting pertinent information. The work then uses standard Weka classifiers [27] and reports training and testing time of several classifiers such as *K*-Nearest Neighbor, Decision Tree, Naive Bayes and Support Vector Machines. Anomaly and change detection are two approaches that are applied for increasing the chance of detecting unknown malware.

2.1.3 Citadel Malware Analysis

The Citadel malware has been chosen as a case study in this work and the reverse engineering process is discussed in Chapter 5. In this section, the related work regarding Citadel malware analysis is presented.

AnhLab has provided a comprehensive static analysis of the Citadel malware [51]. To the author's knowledge, this report is the most complete analysis of Citadel malware, which has been released so far. The process of infection, the structure of the malware binary, and malware's main functionalities and features are explained in detail in this technical report. The report gives valuable insights on the malware and its capabilities, however, the methodology and steps that were taken for reaching the outcomes were not discussed. Also, although it is mentioned in the report that Citadel is remarkably similar to Zeus, the precise quantification of their similarity is not provided. Only approximate resemblance percentages are given without any details. To compare our analysis to this work, we provide a new methodology for reverse engineering malware by adopting clone-based analysis. Following our methodology, we concisely explain the steps that are taken in reverse engineering Citadel and insights that have been obtained through our study. Additionally, by leveraging the tools developed in our security lab, we precisely quantify the similarity between Zeus and Citadel malware.

SophosLabs [52], provided a brief report on Citadel malware. The major enhancements occurred in Citadel comparing to Zeus is explained in high-level and very briefly in this report. No explanation is provided about the process of reverse engineering the malware and how the authors gained those insights. Indeed, this report gives a decent overview about the Citadel malware without digging into the details. CERT Polska [53], also provided a technical report on Citadel malware. Similar to the previously mentioned report, this report is high-level and goes through the main features of Citadel without providing details. The reports mainly provided statistics focusing on the impact of the malware and its geographical distribution. The statistics were gathered based on the traffic to the sinkhole server after the domain take-down. The report by Dell [57] provides

an overview of the Citadel features and provides a lists of Citadel DNS filter domains related to Citadel's DNS poisoning attack.

2.2 Clone-based Analysis

Clone-based analysis is a new approach, which has been recently adopted in malware analysis scenarios. Two types of clone-detection techniques are related to the current work. The first approach is known as binary clone detection, which focuses on comparing two assembly files for finding potential assembly clones [49]. The second approach is called source to binary clone detection [47]. Regardless of the search method, a similarity metric should be utilized for computing the distance between the clones. For this purpose, representative assembly and source feature vectors are generated, indexed, and compared against each other. The clone-based analysis is able to find the previously analyzed code in the disassembly using a repository of source or assembly code. The following sections present the tools and techniques of this analysis.

2.2.1 Source and Binary Code Similarity

Source-to-source similarity can be measured from different perspectives. The research on source code clone detection investigates string-based, token-based, structure-based, and semantic-based approaches for finding similar code [21, 31, 33]. Some techniques are based on direct source code comparison and others are based on indirect methods [35]. One important application of source code comparison is in *plagiarism* detection [34]. In that context, an accurate plagiarism detection system should be robust to code transformations such as variable and parameter renaming,

reordering of class attributes and methods, and slight changes in control flow and loop structures. Generating an intermediary representation of the program code is an important and common step among most of the approaches. We attain this representation by defining feature vectors similar to structure-based techniques for online analysis in *RE-Source*.

Assembly clone detection techniques measure the similarity between two binary files. These approaches use several layers of pre-processing on assembly mnemonics and operands for generalizing the instructions and registers. Similar to the source clone detection techniques, different types of clones (*Types I, II, III, IV*) are defined [17, 23, 49]. Type I clones are know as exact clones and they may differ in whitespace. In type II clones, the syntactic structure is often preserved and the variation is related to identifiers, layout, comments, and types. In Type III clones, certain statements are added or removed from the fragments to further modify the clones. Type IV clones are know as semantic clones. The detection techniques could be based on text search of identical byte sequences or token based analysis of *n*-grams and hash-based fingerprints. Other approaches consider distance metrics, structural similarity, control flow analysis, memory and window-based assembly analysis. Besides, hybrid approaches for the definition, extraction, and processing of clone matching features are also common. The work by Farhadi [48] presents a tool called *RE-Clone* for binary clone detection. As it will be discussed in Chapter 5, this tool is used during the Citadel malware analysis for comparing Citadel versus Zeus. *BinDiff* [61] is a similar tool which is used for finding assembly code similarities. *HBGary Fingerprint* tool [64] is a binay analysis tool for comparing two executable files. It aims to cluster similar binary files into groups based on file attributes and development environment. Besides, new fingerprints can be defined for file classification purposes. Even though most of the features are not at assembly level, it can be used

as a tool for grouping similar binaries prior to the low-level analysis.

2.2.2 Recognizing Libraries and Program Elements

It is desirable that certain malware components such as standard libraries and function calls be identified during the initial phases of malware analysis. Many studies have been carried out on detecting library functions in binary files [24, 30]. Some approaches employ byte-sequence signature matching of function code. Function patterns could be defined as simple as the first n bytes of a function code [24] or in a more abstract format based on data and control flow analysis [19]. The intermediary and semantic program representations are best suited for generating patterns that are used for recognizing program elements. Also, selecting the appropriate features for pattern definition will have a direct impact on the results of the pattern matching process. It should be noted that low-level binary code is machine-dependent whereas high-level code is language-dependent. Thus, generating program patterns based on the intermediate representation is a reasonable approach, since the essential information can be captured and conveyed for effective pattern recognition. Exact and approximate pattern matching techniques are common for this purpose. Even though exact pattern matching methods are accurate, they are ineffective in case of slight byte level discrepancies. On the contrary, inexact or approximate techniques are more robust for finding imperfect and rough matches. The problem of library fingerprinting was investigated in [30] and a method based on semantic descriptors was utilized to identify indirect invocation of system calls using an inexact pattern matching technique. System calls can be considered as interaction points with the operating system and could provide significant information on a binary's behavior. In the malware context, system calls are typically wrapped by other functions hierarchically in order to

hide the underlying functionality. Adopting approximate pattern matching methods is the key for identifying function calls from standard and open source libraries. *IDAscope* [62] is an excellent tool for identification of assembly level features such as standard constants used in cryptographic algorithms. The search can be customized based on user-defined parameters. Furthermore, it provides hints on the Windows API calls used in assembly functions. This tool uses a Python script developed by A. Hanel [60] for classifying functions in groups of API calls. We draw inspiration from this tool and design an extended version, which not only reports on system calls, but also provides insights into the malware functionalities.

Summary

Malware reverse engineering process consists of static, dynamic, and memory analysis steps and many tools and techniques are available for supporting the process at each step. This chapter briefly discussed the process and presented the research related to the main components of the *RE-Source* framework. In the following chapter, the elaboration as well as the design and implementation details of the framework are introduced.

Chapter 3

The RE-Source Framework

This chapter introduces the foundations of the *RE-Source* framework and describe the design and implementation processes. Software reverse engineering is a fastidious task demanding a strong expertise in assembly coding. Various existing tools may help analyze the functionality of a binary file without executing it and an interesting step would naturally be the search for the original source files. The *RE-Source* framework considers the extraction of features in the assembly code so that queries can be triggered to a source repository in a reliable way: either (1) the result is a set of references to the original project files provided they are hosted on the repository or (2) at least some functionalities of the binary file are unleashed. Such an approach is very promising given its proved performances in real assembly code applications. The chapter is structured as follows: In Section 3.2, we present our methodology followed by details of the underlying design and implementation. Two experimental scenarios are described in details in Section 3.3 followed by the conclusion.

3.1 Introduction

Software reverse engineering consists of studying and understanding the process by which a machine-generated assembly-language program has been created by working backwards [3]. If manually writing assembly (ASM) code involves specific programming skills, a compiler automatically converts a high level language such as C into a machine code. The ASM analysis becomes extremely challenging especially if the compiler adds certain optimizations by rearranging the computations, changing or replacing some operations.

Common reverse engineering practices suggest two approaches, namely dynamic and static, with the binary file as the starting point. By dynamic approach (e.g., [10]), we mean isolating the binary file in an application specific environment to model its behavior by execution. Since this does not necessarily reveal all execution flows, debugging tools (e.g., *WinDbg* [16], *Gdb* [5], *Valgrind* [15]) are often associated with this method. As long as only the functionality is targeted, the dynamic approach is acceptable. In other situations, static analysis yields better results and does not compromise the security requirements of the analysis environment.

The first step of the static analysis of a binary file is the disassembly phase. The disassembler (e.g., *objdump* [6] in Linux) is a program considered of invaluable help since it generates the ASM code of the binary file. At this level, mastering the ASM program representation seldom leads to fully understanding the program functionalities. More advanced disassemblers such as *IDA Pro* [59] are meant to simplify the analysis by offering a rich GUI with the program divided in blocks and a program flow graph (PFG). A challenging step further is then to obtain a correct higher level program representation, i.e., the source files. A decompiler (e.g., *Hex-Rays* [12] or

TyDec [14]) could help a lot but since there is not always a one-to-one correlation between the ASM and the source objects, the automatically generated sources may be difficult to follow. For example, it is not simple to detect the definition of object structures in ASM.

3.2 Design Methodology

The input to our process is an ASM file resulted from disassembling the target binary via *IDA Pro*. The specific representation of the ASM, together with its PFG program flow graph, allow us to consider the partitioning of the ASM code in blocks, each corresponding to begin *proc / end proc* where proc stands for procedure. Here is an example of a simple code in C.

```
int sum(int a, int b){
return a + b;
}
```

The corresponding ASM code contains a procedure that we can easily identify by its *name* “sum" (IDA Pro encloses it with the *begin proc* and *end proc* keywords):

```
sum:
push %ebp
mov %esp,%ebp
mov 0xc(%ebp),%eax
add 0x8(%ebp),%eax
pop %ebp
ret
```

We thus consider an ASM file as a set of procedures that are to be individually analyzed by our framework. Each procedure may contain some *interesting features* (see Section 3.2.2) that our tool is able to extract and exploit in order to submit queries to a source repository. The result is (1) either a set of links to pertaining source files referencing the same features, links that we insert as comments in the original ASM file, or (2) the insertion of a comment about the functionality of the current procedure after its local offline analysis (cf. Section 3.2.2).

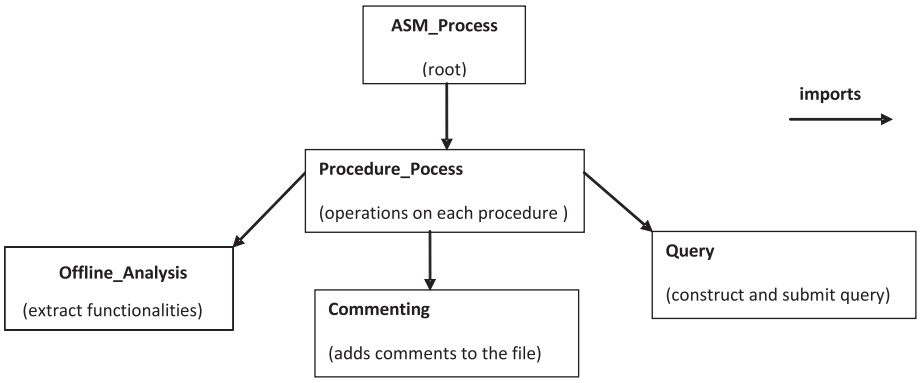


Figure 3: Algorithm Decomposition.

3.2.1 Modules and Algorithms

We adopt a B-Method like notation [1] to abstract and describe the algorithm implemented by our *RE-Source* framework. Fig. 3 captures the RE-Source algorithm decomposition in B-like components. Thus we have identified five modules: (1) *ASM_Process* root machine, which interfaces the user. It imports the (2) *Procedure_Process* machine responsible with processing each ASM procedure: it calls the operations of the (3) *Query* module in order to submit queries to a *set* of code repositories, the operations of the (4) *Offline_Analysis* module, which is in charge of a local analysis to extract the program functionality and also the operations of the (5) *Commenting* module, which adds the pertaining comments to the original file.

***ASM_Process* Module**

Any ASM file is a SET of PROCEDURES. As we can easily depict from Fig. 4, we take as input to our algorithm the ASM_Original_file. It is of type PROCEDURES and remains CONSTANT: these assumptions are captured by the CONSTANT and PROPERTIES clauses. The only variable we introduce is a set of SOME_PROCEDURES among those presented by IDA Pro that the user

```
MACHINE ASM_Process
IMPORTS Procedure_Process
SETS
    PROCEDURES
CONSTANTS
    ASM_Original_file
PROPERTIES
    ASM_Original_file ∈ ℙ(PROCEDURES)
VARIABLES
    Some_Procedures
INVARIANTS
    Some_Procedures ⊆ ASM_Original_file
INITIALISATION
     Some_Procedures := ∅
OPERATIONS
    try_read_procedures(procs) = PRE procs ≠ ∅ ∧ procs ⊆ ASM_Original_file
                                 THEN Some_Procedures := procs
                                 END;
    process = PRE Some_Procedures ≠ ∅ THEN
              VAR F1, F2, p IN
                 WHILE Some_Procedures ≠ ∅ DO
                     ANY p WHERE p ∈ Some_Procedures THEN
                      F1, F2 ← read_features(p); /*from Procedure_Process*/
                      query(F1); /*op.  in Procedure_Process*/
                      analyse_locally(F2); /*op.  in Procedure_Process*/
                      update(p); /*op.  in Procedure_Process*/
                      Some_Procedures := Some_Procedures \ {p};
                     END
                 END
              END
             END
END/*ASM_Process*/
```

Figure 4: ASM_Process Module

chooses to analyze. This variable may be modified by the OPERATIONS, which must always satisfy the INVARIANT. Here, the INVARIANT states that the procedures to be analyzed are part of the original ASM file. In Fig. 3.2, F1 and F2 correspond to features for online and offline analysis respectively.

***Procedure_Process* Module**

For each procedure, there is a phase of ASM *features* extraction, followed by submitting queries to source repositories and by performing a local analysis.

```
MACHINE Procedure_Process
IMPORTS Query, Offline_Analysis, Commenting
SETS
    FEATURES
VARIABLES
    OnFeat, OffFeat, queried, analysed, updated
INVARIANTS
    OnFeat ⊆ FEATURES ∧ OffFeat ⊆ FEATURES ∧ queried ∈ BOOL ∧
     analysed ∈ BOOL ∧ updated ∈ BOOL
INITIALISATION
     OnFeat, OffFeat, queried, analysed, updated := ∅, ∅, false, false, false
OPERATIONS
    F1, F2 ← read_features(p) = PRE p ≠ ∅ THEN
                                /*features extraction from p: to refine*/
                                F1:=OnFeat; /*features for online analysis*/
                                F2:=OffFeat; /*features for local analysis*/
                                queried, analysed, updated := false, false, false;
                            END;
    query(f) = PRE f ⊆ OnFeat ∧ queried = false THEN
               IF f ≠ ∅ THEN
                     submit_query(f); /*operation in Query machine*/
               END
               queried := true;
               END;
    analyse_locally(f) = PRE f ⊆ OffFeat ∧ queried = true ∧ analysed = false
               THEN /*local analysis for functionality extraction */
               /*based on operations in Offline_Analysis machine*/
               analysed := true; append_to_log(f); /*displaying results*/
               END;
    update(p) = PRE queried = true ∧ analysed = true ∧ updated = false THEN
               /*updates after the online query and the local analysis*/
               /*based on operations in Commenting machine*/
               updated := true;
               END;
END/*Procedure_Process*/
```

Figure 5: Procedure_Process Module

We identify these steps in the *process* operation of Fig. 4. The *Procedure_Process* module uses the services of the *Query* and respectively the *Offline_Analysis* modules for the specific *query* and *analyse_locally* operations (Fig. 5). These operations are to be carefully implemented since their

abstract representation cannot contain too many details. The module states only the permitted order in which these operations are called via the PRE-condition clause.

The *process* operation of Fig. 4 considers a last phase of *updating*: the ASM original file remains the same, i.e., *constant*, except for the ASM comments part, which gathers the query results and the local analysis. Therefore, the result of the entire process is the original file updated with comments as we shall see in Section 3.2.2.

***Query* module**

Based on the extracted features in a procedure, its role is to construct and submit queries to a set of source repositories, which are previously known. We could use an instantiated SET of repositories to capture this detail but for the sake of simplicity, we choose to identify each source repository with a natural number in the set 1..`m` where `m` is the number of repositories as depicted in Fig. 6.

Moreover, we also express the following requirement: a real source repository may not be queried too frequently and consequently there should be a mechanism to launch the query to a different *queryable* repository so that the process does not stop. The straightforward way is to introduce a CONSTANT function SEQ_REPS, which gives the *next* source repository to query. Implementing this is based on the observation of some query interval slots for each real repository and by defining thus an order of passing from one repository to another. Then `Queryable_Reps(r) = true` means that the `r` repository can accept a query. This variable is modified in the implementation of the submit_query operation.

We do not give the B notation of the *Offline_Analysis* and Commenting modules because their

```
MACHINE Query
CONSTANTS
    n, SEQ_REPS
DEFINITIONS
    Repositories == 1..n
PROPERTIES
   n ∈ ℕAT1 ∧ SEQ_REPS ∈ Repositories → Repositories
VARIABLES
    Queryable_Reps
INVARIANTS
   Queryable_Reps ∈ Repositories → BOOL
INITIALISATION
    ran(Queryables_Reps) := true
    /* all repositories should be queryable at the beginning*/
OPERATIONS
    submit_query(F) = ANY r WHERE Queryable_Reps(r) = true THEN
            /*submit query*/
            Queryable_Reps(SEQ_REPS(r)) := true;
            END
END/*Query*/
```

Figure 6: Query Module

operations proved to be more challenging to implement at low level. The *append_to_log*() operation is meant to save the execution steps in a log file at runtime.

3.2.2 Implementation Details

If a B-like specification is useful to examine the possible flows and to define the operations preconditions and the invariants they have to meet, the validity of the low level implementation is generally asserted using normal techniques such as testing and peer code reviewing.

Thus *RE-Source* program implements the algorithm as a Python IDA Pro plug-in. It is worth mentioning that, unlike the *RE-Google* plugin [9], our extended version does not rely on the Gdata framework [8], nor does it utilize Google Code [7] as the only search engine for accessing code repositories. Instead, it possesses a built-in query processing engine and parsing mechanism for

handling request/response messages. Furthermore, it supports multiple search engines and it provides a framework for adding new code repositories with only a few lines of code. Also, the program makes use of an interleaving time optimization technique for managing multiple search engines. Despite the large number of request/response messages, it honors the required time delays between consequent messages without prolonging the processing time.

In terms of extracted *interesting features* from the ASM code, *RE-Google* considers only three features, namely constants, strings, and function imports. In contrast, *RE-Source* is able to extract four types of features for online analysis and query building: (1) immediate values of operands, (2) imported libraries and function calls, (3) exported functions in DLLs, and (4) strings values. Besides, it considers eight features for offline analysis. For each function, we extract information about its stack frame: (1) number of instructions; (2) size and number of local variables; (3) size and number of arguments; (4) size of saved registers; (5) function flags; (6) function addresses (begin, end, return); (7) function prototype (type of input and output and calling convention); (8) calls to low level system functions (malware dictionary). Moreover, variable scopes (local/stack-based or global/memory-based) and simple data structures (single variables or *structs*) are highlighted for the reverse engineer as well.

Moreover, the program adds better result-handling techniques than *RE-Google* and an offline functionality analysis engine. In many situations, online results may not be available due to the lack of extracted features, obfuscated or hard-coded procedure, use of complex and non-standard algorithms, etc. Therefore, the offline analyzer is of great benefit for revealing the overall functionality of a portion of assembly code. It has an expandable dictionary of common functions in Windows API along with a programmer-friendly description of each function.

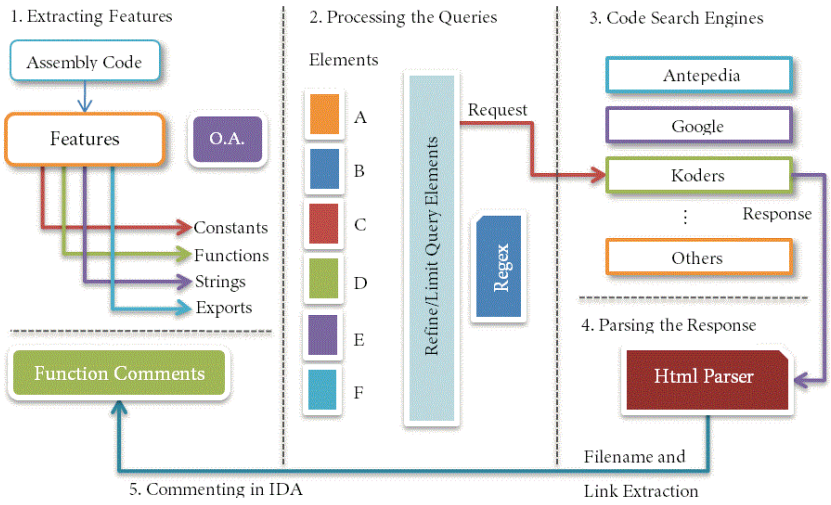


Figure 7: Program Execution Flow

Program Execution Flow

As illustrated in Fig. 7, there are five main modules in the Python program for handling tasks related to features, queries, repositories, parsing and commenting. Except for the Code Search Engine (3), these modules have a counterpart in the algorithm blocks of Fig. 3. The *RE-Source* program interacts with *IDA Pro* API for getting a list of available procedures in the disassembly, getting function addresses and names, as well as adding comments to the file.

The execution flow starts in the main function of the script where the initialization of variables and execution-time calculation is done (*Initialize*(*RESrc_Vars*)). Then, the script checks a variable (flag) to determine whether the search should be performed on a specific function or on all the extracted functions from the disassembly (*RESrc*(*asm_function_list*)). In the first case, the user highlights a specific function for search and in the second case all the functions are taken into account. In the next step, the *RESrc* function counts the total number of available procedures

and prepares a loop for analyzing each item. Then, a function will be called for extracting four types of features namely: constants, imported libraries, exported libraries and string values from the disassembly. The output of the *Extract_QFeatures()* function is a potential list of features that could be used for building a general query. This list will be refined in several layers before building a specific query. Next, the features for offline analysis are extracted using *Extract_OAFeatures()* function.

```
feature_qlist ← Extract_QFeatures( asm_function )
feature_oalist ← Extract_OAFeatures( asm_function)
            // OA for Offline Analysis
      feature_list_func_i = [ f_i1, f_i2, ..., f_in ]
  refined_qlist ← Refine_GQuery( feature_qlist )
```

The initial purpose of the *Offline Analysis* (O.A.) module was to compare a function with a list of known Windows API functions in order to get a simple statement about the functionality and prototype of the procedure under analysis. In Chapter 4, we show how to extend this mechanism for providing an advanced mapping. Also, this module assists the reverse engineer by highlighting the variables and their scope.

The purpose of the *Refine_GQuery()* function in the refining process is to filter out certain characters from the feature list to prevent problems with search engines queries. For instance, the search engines may not allow characters such as "%", ",", "’" as part of query string to prevent SQL injection. Therefore, the output query is safe for submission to code search engines. However, the user can define what characters are blacklisted by adding 'badkey': 'value' pairs into the "BlackDict" dictionary. For instance, the keys in the following dictionary are simply replaced with the '' character, which is equivalent to removing them from the search string.

```
BlackDict = {'%d':'', '%s':'', '\\':'', '%1':'', ...}
```

Also, this function encodes and prepares the list for the next steps of specific query building:

```
base_query ← Generate_Query( refined_qlist )
```

At this step, we have a base query that can be further encoded for particular search engines. Search engine-specific prefix and suffix will be added to each base query to build a standard query. The following functions are examples of query building functions for three major code search engines.

```
final_queryKoders ← Build_Koders_Query( base_query )
   final_queryGCS ← Build_GCS_Query( base_query )
final_queryKrugle ← Build_Krugle_Query( base_query )
```

The next step is to submit the query and get the response for each respective search engine. The order of query submission and response extraction is important for time optimization. Usually there must be a time delay between two subsequent requests to a search engine (SE). The program uses an interleaving technique for managing the query submission and for saving processing time (an example is hown in Fig. 8).

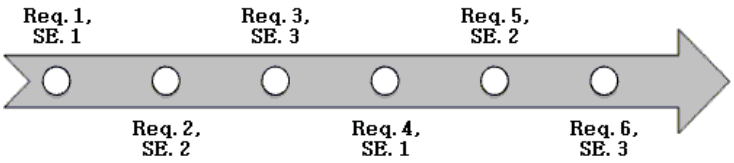


Figure 8: Search Engine Time Interleaving.

For each query, a request is made and the response page is received in HTML format.

$$\texttt{html\_page}_j \leftarrow \texttt{Fetch\_Response(final\_query}_j \texttt{)}$$

After getting the page, a call to the local parsing function will be made to extract relevant information based on a predefined regular expression statement for each search engine. Then, the matching *filenames* and *URLs* are extracted and stored in a dictionary. The `Parse_Page()` function receives a web page that contains tabular information about matched projects and files. Then, it searches the page for instances of data which match the regular expression and return the values as output.

```
dictionary_list ← Refine_Results( Parse_Page( html_page ) )
```

The results are processed and duplicate results are removed from the list. Also, based on the search engine rankings, the best matches are selected and given priority. Lastly, the comments are updated to reflect the online search results.

$\texttt{function\_comment} \leftarrow \texttt{Update\_Comment}_{function\_k}\texttt{(refined\_dictionary\_list)}$

In the next section, we present a practical application of the *RE-Source* framework in analyzing an open source software.

3.3 Experimental Results

We have adopted the *PreciseCalc Project* [11] given that both the sources and binary files are available on SourceForge and Koders (*http://www.koders.com*) as a code search engine. As an input to our *RE-Source* IDA Pro plugin, we use the assembly file resulted from disassembling the *PreciseCalc* binary. There are 533 assembly procedures and we choose to analyze the total disassembly.

1	* Analyzing function 4 [sub_4013A0] @ [0x4013a0] Constants: ['0x13880', '0xf4240', '0x4c4b40', '0x493e0'] Strings: [] Imports: set([]) list: 0x13880 0xf4240 0x4c4b40 0x493e0 * Looking for an exact match * * Looking for a rough match based on constants *
2	* Analyzing function 127 [sub_407F40] @ [0x407f40] Constants: [] Strings: ['"arctan, arccot from 1i or -1i"', '"argtanh,argcoth from 1 or -1"'] Imports: set([]) list: "arctan, arccot from 1i or -1i" "argtanh,argcoth from 1 or -1" * Looking for an exact match * * Looking for a close match based on strings *
3	* Analyzing function 334 [sub_417B20] @ [0x417b20] Constants: ['0x80000001', '0x80000001', '0x80000001'] Strings: ['"Software\\Petr Lastovicka"', '"Software\\Petr Lastovicka"'] Imports: set(['RegDeleteKey', 'RegOpenKey', 'RegCloseKey', 'RegQueryInfoKey']) list: 0x80000001 0x80000001 0x80000001 "Software\Petr Lastovicka" "Software\Petr Lastovicka" RegDeleteKey RegOpenKey RegCloseKey RegQueryInfoKey * Looking for an exact match * * Looking for a close match based on strings * * Looking for a rough match based on constants *
4	* Analyzing function 375 [sub_41B4A0] @ [0x41b4a0] Constants: [] Strings: ['"sin"', '"cos"', '"tan"', '"tg"', '"cot"', '"cotg"', '"asin"', '"acos"', '"atan"', '"atg"', '"acot"', '"acotg"', '"arcsin"', '"arccos"', '"arctan"', '"arctg"', '"arccot"', '"arccotg"', '"arc"'] Imports: set(['strnicmp']) list: "sin" "cos" "tan" "tg" "cot" "cotg" "asin" "acos" "atan" "atg" "acot" "acotg" "arcsin" "arccos" "arctan" "arctg" "arccot" "arccotg" "arc" strnicmp * Looking for an exact match * * Looking for a close match based on strings *

Figure 9: Feature Extraction (excerpt from the log file).

The `Extract_QFeatures()` function is able to extract features from 67 procedures. If there are at least two elements in the *Imports list*, or the joint set of *Constants* and *String List* is non-empty, then the script would try to find an *exact match* by concatenating all the elements. This is an ideal situation where the query would be expressive enough in terms of number, and the type of elements. If no exact match is found then the search would be based on the strings inside the binary. If the length of String List is larger than one, then the search query will be built by concatenating the String elements. Finally, if there is at least one element in the Constants List but the results set is empty, the script will perform the search by building a query based on the

Func. no.	Function ID @ Address	Source Code Link	Match
70	[sub_406800] @ [0x406800]	complex.cpp	100%
146	[sub_409620] @ [0x409620]	lang.cpp	100%
159	[sub_40A1E0] @ [0x40a1e0]	matrix.cpp	100%
261	[sub_4119B0] @ [0x4119b0]	parser.cpp	100%
334	[sub_417B20] @ [0x417b20]	preccalc.cpp	100%

Table 1: Identified Source Codes.

concatenation of constant elements.

The conditional rules for defining each case can be altered based on the application under analysis and the number of available elements in the extracted lists. Generally, there are more elements in each list when the application makes use of standard Windows libraries.

Fig. 9 shows four examples of interesting features extracted by the `Extract_QFeatures()` function. In the first case, the search is merely performed based on the *constants*. Example 2 shows a situation in which only *string* information is available. No import lists are detected for the first two cases. Conversely, in examples 3 and 4, sets of *imported function* names are included in the search. The original *PreciseCalc* project can be accurately identified by submitting a query containing portions of the strings in example 3. Even if an exact match is not found, *RE-Source* will try to find a close or a rough match based on a combination of features. New rules can be defined for combing the extracted features, depending on the analysis scenario.

RE-Source was able to detect several references to each source file in the original project. The *PreciseCalc* application includes functions that handle text editing, GUI processing, timing and registry modification, alongside the arithmetic, statistical, geometrical and other math-related functions.

Table 1 shows sample results of the identified C++ source code. The identified links and filenames

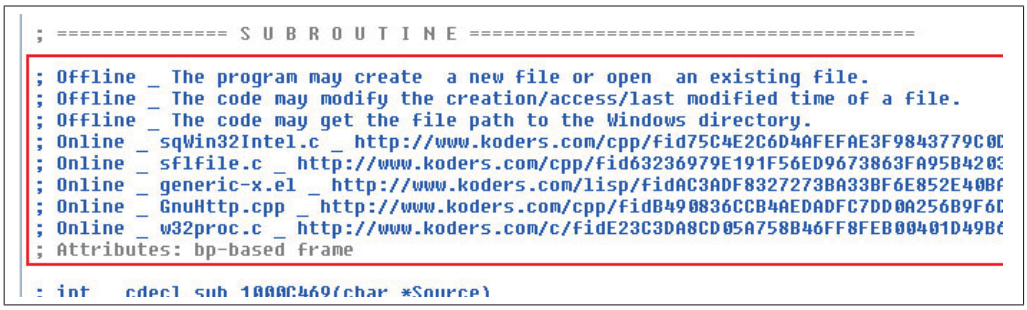
are inserted directly in the assembly file. There are several references to the main “preccalc.cpp” file. For instance, the functions at addresses: `0x417b20`, `0x41b2d0`, `0x4190c0`, `0x419ab0`, `0x41a2b0`, `0x41b4a0`, `0x41be70` and `0x41c1f0` were referencing the main C++ file in the project. Fig. 10 shows one of these references. Besides, it has detected several math-related functions in the disassembly. The script has generated a comprehensive execution log that is self-explanatory and describes the analysis process. Even though the initial version of *RE-Source* did not include heuristic query processing techniques, it was able to detect more than 70% of the original source files with an accuracy of 100%. We had access to the original source code of the project. Therefore we were able to validate the accuracy of the results manually. Also, the script is useful for gaining insight into the functionality of the assembly file.

```
.text:0041B2D0 ; =============== S U B R O U T I N E =======================================
.text:0041B2D0
.text:0041B2D0 ; Online _ preccalc.cpp _ http://www.koders.com/cpp/fidC00F18FC54602A45811CF096F055D2
.text:0041B2D0 ; Attributes: bp-based frame
.text:0041B2D0
.text:0041B2D0 sub_41B2D0      proc near               ; CODE XREF: sub_4099D0+55↑p
.text:0041B2D0                                         ; WinMain(x,x,x,x)+312↓p
.text:0041B2D0                 push    ebp
.text:0041B2D1                 mov     ebp, esp
.text:0041B2D3                 call    sub_418EE0
.text:0041B2D8                 call    sub_4190C0
.text:0041B2DD                 call    sub_4188D0
.text:0041B2E2                 push    offset aPreciseCalcula
.text:0041B2E7                 push    1F8h
.text:0041B2EC                 call    sub_409440
.text:0041B2F1                 add     esp, 8
.text:0041B2F4                 mov     off_425710, eax
.text:0041B2F9                 mov     eax, off_425710
.text:0041B2FE                 push    eax
```

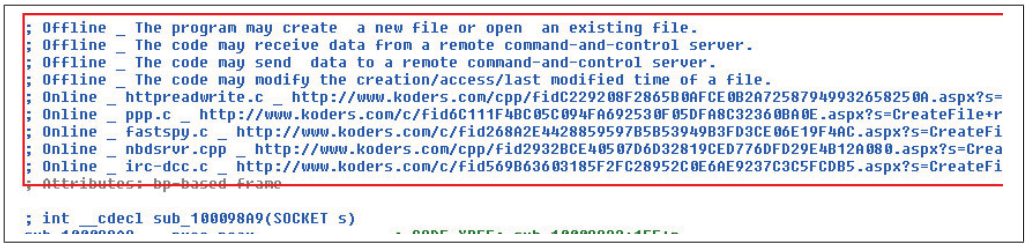
Figure 10: Identified “precalc.cpp" File @ 0x41B2D0.

Concerning the offline analysis module, the current implementation includes a dictionary of common Windows APIs, which includes a brief description of each function. This dictionary was built with malware analysis in mind. Therefore, it includes about 200 of the most common

kernel and user level functions known to be used by existing malware [41]. The offline analysis comments are appended automatically based on the type and classification of system calls in the each assembly function.



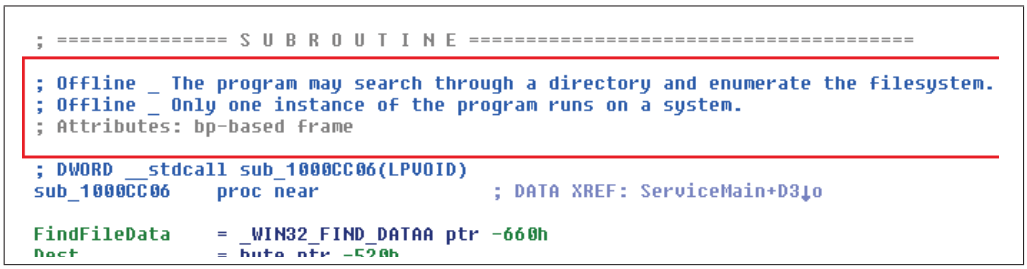
(a) Routine involving file I/O



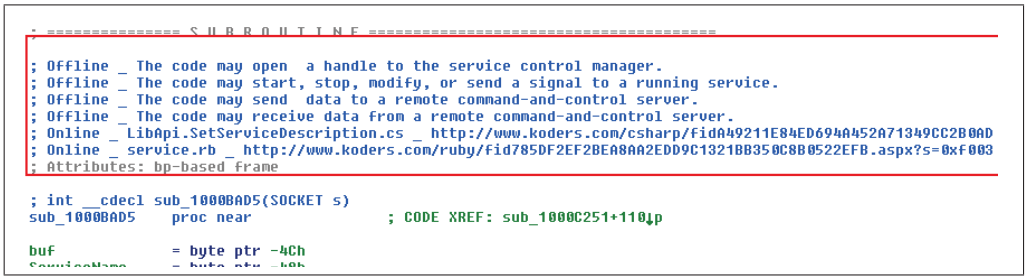
(b) References to some networking services

Figure 11: Examples of the Comments for File I/O and Network.

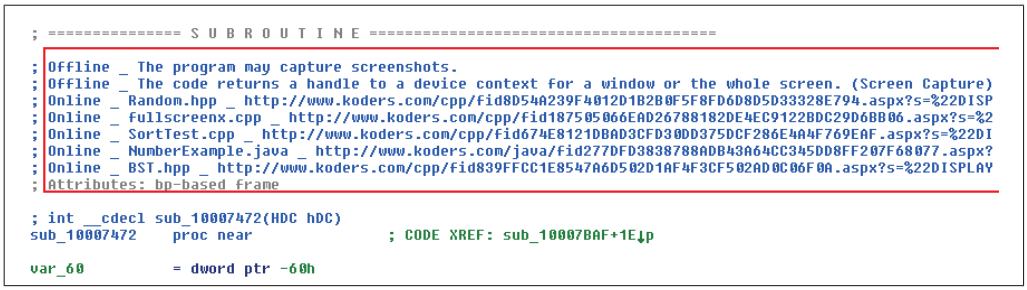
In our second scenario, we run *RE-Source* on several malware disassemblies. The offline analyzer helps the reverse engineers to understand network connectivity and data gathering functionalities of malware by adding relevant comments. In Fig. 11, there are several routines of malware performing file I/O operations and communication with a remote command and control server. In cases where *RE-Source* returns results from both the online code repositories and the offline analyzer, an emergent consistency is observed. As an example, Fig. 3.12(c) depicts a portion of assembly code that is capturing the screen and saves it to a file to be remotely transmitted. As can be seen, *RE-Source* gives reliable information in both offline and online comment sections.



(a) Offline analysis only



(b) References to system services



(c) Routine revealing screen capture functionality

Figure 12: Examples of the Final Outcome.

Discussion

A side by side comparison between outputs of *RE-Google* and *RE-Source* was not possible because the underlying search framework of *RE-Google* had been deprecated. In other words, *RE-Google* is not functional anymore. RE-Source takes an intra-procedural approach to extract features and builds queries. It could be argued that an inter-procedural approach could improve the accuracy of

the online analysis. However, the search engines provide limited commands for executing logic-based queries and some of them do not provide direct APIs to their repositories. Adopting a heuristic query building algorithm that tries different elements in the query string and selects the best match could improve the accuracy of the identified online projects. As to the accuracy of the offline analysis, it clearly depends on the number and selection of the functions defined in the dictionary. In a situation where we have results from both the online and offline analyzers, the reverse engineer would have the maximum information. This happens when programs make use of standard libraries such as VCL or MFC. In other cases, there might be no results from the online module, this happens when malware authors use non-standard components or they use certain wrappers around standard system calls. Also, they might use non-standard low-level kernel functions for performing simple I/O operations.

3.4 Summary

Software reverse engineering is a complex task. Applications like, malware analysis can benefit from a framework that is able to provide information about the matching between open source and assembly code. In this chapter, we established a framework to develop such a tool that exploits features existing at both the source and the assembly file levels. Based on these features, queries are triggered on certain online repositories used by the developers' community. If there is no query result, the tool is still able to provide some information about the functionality of a portion of the assembly file by a local offline analysis. The reverse engineer's task is thus greatly simplified. In the next chapter, we provide an elaborate discussion on the extension of offline analysis.

Chapter 4

Design Methodology for RE-Source Extensions

A design methodology is presented to enhance the process of assembly to source code mapping based on incremental learning of feature vectors. Using a semantically consistent path in a customized code repository, reference functionality patterns are generated, which are later used for classifier training. Unforeseen feature vectors are extracted from all functions in malware disassemblies and compared against the base classes. The result of the comparison is either a functionality tag or a suggestion for manual review. In situations where a classification decision cannot be reached, the reverse engineer's feedback is treated as a new learning sample. Over time, the performance of the system is improved through an active learning mechanism. The proposed methodology has been implemented as an extension to the offline analyzer of the *RE-Source* framework.

In the previous chapter, we introduced the *RE-Source* framework. The proposed methodology to extend the framework is presented in Section 4.2. In Section 4.3, we discuss the experiments

and report the results. Finally, conclusions are made in Section 4.4.

4.1 Introduction

Malware analysis is a critical area in computer security, which has gained unprecedented attention due to global malware threats. The complexity of malware attacks has increased rapidly and smarter solutions are required to keep up with the emerging threats. Many studies have focused on malware detection techniques and different approaches have been proposed to address the problem [26, 39]. As opposed to traditional signature-based detection methods, data mining techniques and tools have been shown to be effective for the detection and classification of new malware [38, 40]. In malware analysis scenarios, tagging a suspicious binary as malicious or benign is insufficient. More specific information is required on the real intent and inner workings of the malware in a timely manner.

During the initial phases of a malware analysis process, efforts are made for gaining essential information about the potential capabilities and purpose of the malware sample. These information will help in recognizing the target of suspicious code and guide the mitigation process of infected systems [32]. The investigation process entails a series of static code analysis steps followed by dynamic monitoring of malware for dissecting the local and network-related behavior. Although static and dynamic analysis tools are widely available, an in-depth and time-consuming process of reverse engineering must be followed before any conclusions can be drawn on the potential functionality of a malware. Thus, advanced techniques and tools that accelerate the reverse engineering process are needed.

The primary objective of this chapter is to present a methodology together with the underlying

techniques that can be used to obtain the malware analysis results faster and more accurately. Its output is in the form of human-readable statements and can be used in investigation reports. The present work is an extension to our *RE-Source* framework [47]. We contribute by extending the offline analyzer module for active/incremental learning of code functionality based on reference patterns. A local code repository is introduced, which organizes source code based on a set of functionality topics related to common building blocks of malware components. Each category of malware functionality is referenced hierarchically by a path in the repository, which is semantically related to the main topic of the path. The source programs that are referenced in each path share the domain and context of the topic's functionality and have some features in common. Feature vectors are then generated for representing source level information and the corresponding programs are built in order to generate the binary version of source level features. Furthermore, by having access to source level and binary-level features of the same functionality, a mapping is formed and learned by a classifier. Once a new binary-level feature vector is encountered, the system is able to find the closest class of functionality based on the recognized reference patterns. This process is applied incrementally on every function of a malware in its disassembly form. The outcome of the classification phase is either one or more class tags associated with the potential functionality of the code, or a message that suggests a manual review by a reverse engineer. Following the manual review, the system will use the feedback from the reverse engineer along with the representative feature vectors of the function under analysis as a new learning pattern, and will use it to improve the accuracy of the supervised learning process. Over time, the system will improve the mapping between reference patterns and the semantic functionality classes.

In our malware analysis approach, we utilize data mining and machine learning techniques

for a different purpose. Rather than using them for classifying files as malicious or benign, we apply them for gaining insights into function semantics, i.e., for getting a high-level picture of the malware code functionality by incrementally learning joint feature vectors and functionality tags. Also, we integrate an active learning paradigm which is meant for improving the mapping between representative features of source and assembly by getting feedback from the reverse engineer in malware analysis scenarios.

4.2 Methodology

In this section, we present the proposed methodology and formalize each step of the process using set theory notation. The algorithm comprises two parts: (1) Algorithm 1 summarizes the process of feature extraction, program generation and classifier training for the base reference classes, and (2) Algorithm 2 describes the process of matching an unforseen binary feature vector against reference classes. The classification decision is reached by testing the classifiers with extracted features from the target file.

4.2.1 Approach Overview

In our approach, we focus on creating functionality-oriented reference patterns for source code and their corresponding assembly format. In the initial phase, we utilize a number of standard libraries and open source code repositories for building a searchable index of exported function prototypes. For each indexed function, we generate a source-level signature based on the function name, typing information, arguments list, return values, and domain information. These signatures are maintained for extended library recognition.

In the second phase, the following steps are performed in an incremental fashion: First, we define reference base classes of code functionality from the perspective of operating system interaction. Each class encapsulates relevant system calls and data structures. This step is then followed by a compilation phase of a small program that imports and calls functions from the reference class. The output program is dependent on user-specified configurations and compilation settings. Subsequently, a binary file and the resultant disassembly of the program are built and stored locally. Secondly, an intermediary representation of joint feature vectors is synthesized, which takes into account information from high-level source code, signatures, and the low-level disassembly.

The aim of the third phase is to generate an API-based functionality dataset using the crafted feature vectors and the class tags. Each data entry in the dataset corresponds to attributes of a specific program that implements representative methods of the reference class. The dataset is used in a supervised learning process in which different classifiers are trained with correlational data of reference tags and feature vectors. Furthermore, the training data are eventually updated according to the feedback received from a reverse engineer, as an active learning process. The outcome of this phase is a classifier capable of recognizing new code based on its potential functionality.

After preparing the dataset and training the classifiers, a new feature vector can be categorized into one of the predefined classes. This step is done by measuring the similarity between the new program instance and the existing patterns. The new feature vector characterizes the extracted features from a program, or a subset of the program in assembly format. The new analyzed sample will be added to the training set later. As an optional step in the algorithm, a reverse engineer could provide feedback on the quality of the classification after analyzing the subroutines manually. This feedback would be used to adjust the classification output for future samples and could be

considered as reinforcement feedback.

4.2.2 Building a Customized Local Code Repository

Many code repositories group source files by project. Each project contains different source files in several levels that together form the building blocks of a software application. A makefile configuration can be defined for a project path to guide the compilation, linkage, and binary generation.

We take an alternative approach for building a local code repository. Instead of storing files by project, the files are grouped into logical fragments according to the code functionality. The reason behind this decision is to generate a collective feature vector for each representative group and train a classifier in later steps. The intuition is that fragments with a mutual purpose have some features in common. The similarity could be due to the domain/context information in program identifiers/comments, structural similarity in data type definitions, function calls from standard or common libraries, control or data flow information, or other lexical attributes of a programming language [34, 35]. For instance, consider having a group of code written in C++ that points to the socket network connections for creating multi-threaded TCP servers, cf. Figure 13. Essentially, this group will have common features for structures, callbacks and system calls of socket APIs. Likewise, cryptographic algorithms that perform the same type of encryption or decryption have some features in common. The similarity could be related to the constants for key definition and initialization or to the implementation building blocks.

The code classification process can be done in several ways such as supervised, conducted by a reverse engineer, or unsupervised clustering based on frequency-based attribute rankings [25]. This process could be thought of as a dataset generation procedure. A local code repository with

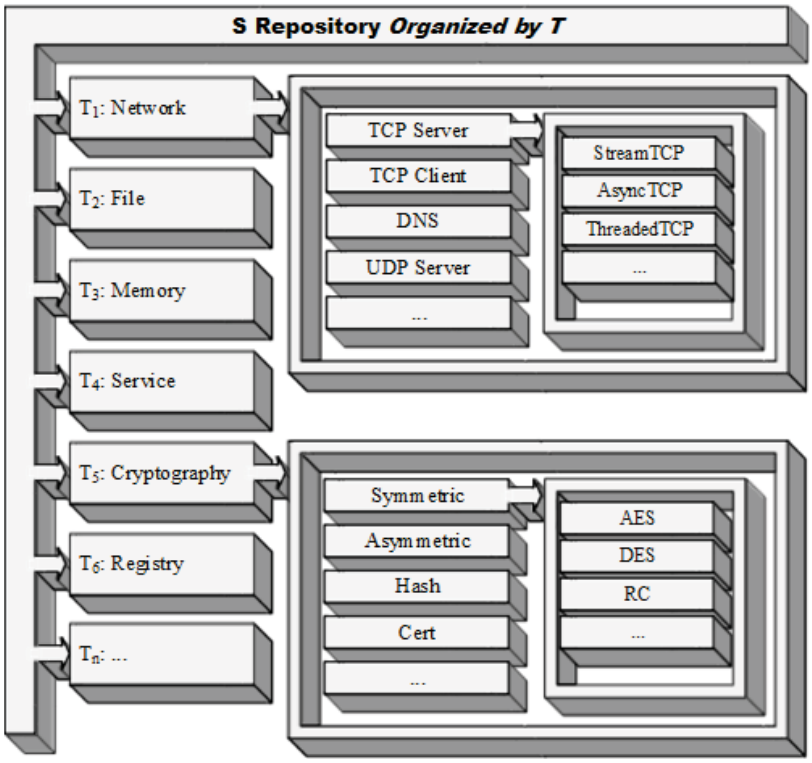


Figure 13: Sample Structure for the Code Repository.

indexing capabilities is designed for storing the initial files and it is incrementally updated throughout the execution of our system. A formal description of the repository is presented in the following sections. The used notation come from set theory and the required operators are defined in Table 2 in which x and y are variables, s, t, u and v are sets, while q and r are *relations* [1]. For instance, $r \in s \leftrightarrow t$.

4.2.3 Source-Level Feature Vectors

Let $\mathcal{S}$ be a source code repository structured by a finite set of n topics. The topics can be considered as labels (tags) for different categories of program files, i.e., $T_n = \{\Gamma_1, \ldots, \Gamma_n\}$. Each item Γ_i is

Operator	Definition	Condition
$\leftrightarrow$	$\mathbb{P}\ (s \times t)$	
r^{-1}	$\{y, x \mid (y,x) \in t \times s \wedge (x,y) \in r\ \}$	$r \in s \leftrightarrow t$
dom(r)	$\{x \mid x \in s \wedge \exists\, y\ .\ (y \in t \wedge (x,y) \in r)\}$	$r \in s \leftrightarrow t$
ran(r)	dom(r^{-1})	$r \in s \leftrightarrow t$
q;r	$\{x,z \mid (x,z) \in s \times u \wedge \exists\, y\ .\ (y \in t \wedge (x,y) \in q \wedge (y,z) \in r)\}$	$q \in s \leftrightarrow t, r \in t \leftrightarrow u$
id(s)	$\{x,y \mid (x,y) \in s \times s \wedge x = y\}$	
$u \lhd r$	id(u);r	$r \in s \leftrightarrow t, u \subseteq s$
$r[v]$	ran($v \lhd r$)	$r \in s \leftrightarrow t, v \subseteq s$
$s \nrightarrow t$	$\{r \mid r \in s \leftrightarrow t \wedge (r^{-1};r) \subseteq \text{id}(t)\}$	
$s \rightarrow t$	$\{f \mid f \in s \nrightarrow t \wedge \text{dom}(f) = s\}$	
x $\mapsto$ y	x, y	

Table 2: Basic Operator Definitions (B-Method)

defined to be a set of nested subtopic hierarchy of depth *d* in the form of $\tau_{i_1} \subset \tau_{i_2} \ldots \subset \tau_{i_d}$ such that τ_{i_1} represents the most specific, and τ_{i_d} denotes the most generic topic. Moreover, the repository comprises a set of paths called $P = \{\rho^1, \ldots, \rho^k\}$, which jointly addresses the total logical storage space in $\mathcal{S}$. The code repository hosts a set of source files, SF (the folders are also files). Each level in the hierarchy corresponds to a logical location in the repository. We denote each file as f_L<i, ρ, τ>, where L shows the language (e.g., C++), i is a unique index, ρ is the file path (such that $\rho \in SF \leftrightarrow SF$ and $\rho \cap \rho^{-1} = \emptyset$) and τ is a topic tag, given to the file. The language L is made of a set of keywords that is denoted as $\sum_L = \{\omega_1, \ldots, \omega_n\}$ and a set of user-defined identifiers. Each program can be considered as a representation of concepts, which are stated in lexical terms using language keywords and identifiers. Also, we assume that the path ρ is addressed by a given namespace, N_ρ. If SN is the set of all namespaces (e.g., *Sys*, *Net*, *TCPSrv*, are members of SN), then $N_\rho \in SN \leftrightarrow SN$ (with $N_\rho \cap {N_\rho}^{-1} = \emptyset$). For instance, N_ρ = {(System$\mapsto$Net),(Net$\mapsto$Sockets)} captures the `System::Net::Sockets` C++ namespace. To simplify the notations, we denote: $\{(\alpha_1 \mapsto \alpha_2), (\alpha_2 \mapsto \alpha_3), \ldots, (\alpha_{m-1} \mapsto \alpha_m)\}$ in a more compact way as $[\alpha_1.\alpha_2 \ldots \alpha_m]$. We also introduce a function

ls, which takes a path and returns the set of files, which are included in that path, i.e $ls[\{\rho\}] \in \mathbb{P}(SF)$ where $\mathbb{P}$ stands for the power set.

We call a path ρ to be *semantically consistent* with regards to the topic τ, if there is an agreement (consistency) between the topics of files, and topics (labels) of the namespace. In other words, the file topics and path topics come from the same domain. We assume that this condition is true for all files and levels in the namespace. We denote such path as ρ_ψ. Implicitly, this path forms a concept hierarchy.

Source files written in one language, which are grouped under the same topic and path, are considered to have a semantic connection. In other words, given a *semantically consistent path* ρ_ψ, a representative feature vector $\overrightarrow{V}_{\rho\psi}$ can be generated for capturing pertinent source-level features associated with domain ρ_ψ such as classes, structs, interfaces, enumeration, delegates, attributes, methods and properties.

We define $\lambda \in ls[\{\rho_\psi\}] \rightarrow \overrightarrow{V}_{\rho\psi}$ as a feature selection function which takes the set of files under a semantically consistent path as input and builds a set of *n*-dimensional feature vectors $\overrightarrow{V}_{\rho\psi} = <v_1 v_2 \ldots v_n>$ in which merely the *n* most significant attributes are retained with regards to the topic. Latent semantic analysis (LSA) is a technique for extracting a set of concepts from documents and finding the associations between the terms in a domain [25]. Latent analysis models can be used for clustering related concepts together and for reducing the size of data. In this context, LSA can help with the identification of significant attributes of a topic based on the term frequencies of the domain. The above defined λ utilizes a ranking functionality based on LSA and a parsing mechanism $T \in SF \nrightarrow T_n$ as a partial function for generating the representative feature vector for a given file.

4.2.4 Source-Level Function Signatures

We define $FP \in arg_t \leftrightarrow ret_t$ to be a function prototype that describes the input and output parameters and types of each function. In this notation, arg_t is the list of input arguments and their types t. Similarly, ret_t is defined as the list of return values and their corresponding types. Each item in the list is represented using a key/value pair.

The parsing function $Par \in ls[\{\rho_\psi\}] \nrightarrow FP$ is defined to take as input the source files in path ρ_ψ and generate a list of function prototypes for them. This list contains symbol names and it is used for generating a unique signature by applying two hash functions h_1 and h_2 on prototypes and symbolic elements referenced in the body of each function. Signatures are compiled for all items in the path by calling a signature maker function $Sig \in FP \rightarrow sig$ (where sig is a set of signatures). This process is applicable for generating signatures from shared libraries.

4.2.5 Making Programs with Chosen Components

Let $\mathcal{CL}$ be a compiler that generates a set of programs Prg from the set Src of source files in a development environment $\mathcal{Dev}$. We denote such mapping by $CL_{Dev} \in Src \rightarrow Prg$, which abstracts object generation and linkage as well. Also, let $\mathcal{DC}$ be a disassembler that maps Prg to an intermediary set of files $ASrc$, which is a different representation of the original Src files. We present this function as $DC \in Prg \rightarrow ASrc$. The mapping relation m_{CL} between $ASrc$ and Src ($m_{CL} \in ASrc \leftrightarrow Src$) depends on the compilation settings, development environment, and configurations. By taking a reverse engineering approach and observing m_{CL} for n selected programs $\{Prg_1, Prg_2, \ldots, Prg_n\}$ with chosen internal components from $\overrightarrow{V}_{\rho\psi}$ and function prototypes FP, one would be able to unleash the fingerprints of the $\mathcal{CL}$ and gain information about development settings of $\mathcal{Dev}$. If

Function				Address				Arguments				Calls				Stack Frame				...
ID	Name	Blocks	Chunks	Start	End	Length	Flags	Number	Size	Type	Value	From	To	API	Library	Register	Size	Return	Conv	...
																				...

Table 3: Portion of the Assembly Feature Vector.

the programs Prg_i are expressive enough in terms of representative $\overrightarrow{V}_{\rho\psi}$ features, they can be used for creating reference patterns of semantic program functionality in *ASrc* form. Even though there are virtually unlimited number of programs that can be written to perform a similar task, they can be considered similar up to some level of abstraction. A properly generalized reference pattern can capture a high-level picture of the code.

4.2.6 Binary-Level Function Patterns

Analyzing functions in assembly format is a challenging task. Marking the correct boundary, scope and range of each assembly routine is usually the first problem to solve. Also, distinguishing between the code and data is equally important. It is assumed that these steps are handled by the disassembler using a robust heuristic technique. Thus, we focus on generating patterns from the disassembly. It is important that library and standard functions be separated from non-standard ones during the analysis. Several types of binary-level signatures are considered for matching source-level information with a binary file. First, a *byte pattern signature* $B_{sig_n} = (b_0 b_1 \ldots b_{n-1})$ is defined as a sequence of bytes of length *n* in the binary format. This pattern is mostly used for identifying an exact match related to crypto signatures. An example in hex format follows:

$B_{SHA_{256-PKCS}}$ = \x30\x31\x30\x0d\x06\x09\x60\x86\x48\x01\x65

\x03\x04\x02\x01\x05\x00\x04\x20

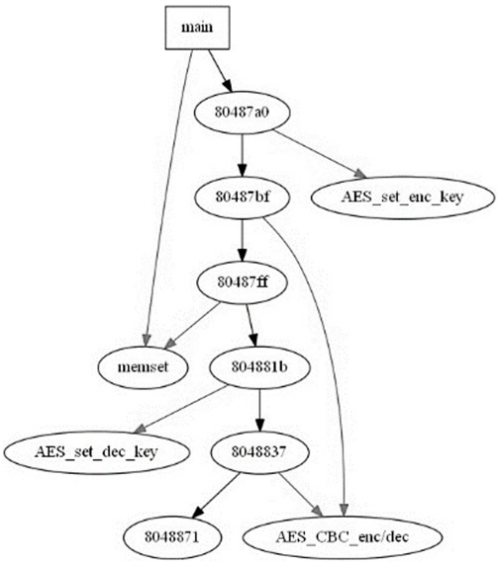


Figure 14: CFG of an AES-based Encryption Function.

In case of an obfuscated code, byte patterns might not be effective due to the lack of static features but the system can set a flag for the obfuscated section. Secondly, a binary feature vector $\overrightarrow{V}_{B_f} = <v_0 v_1 \ldots v_{n-1}>$ is created by fingerprinting each function in the assembly form. Table 3 shows parts of the features that are used for building $\overrightarrow{V}_{B_f}$ when analyzing the functions in the disassembly form. Each feature v_i can hold values such as constants, strings, library imports, function names, calls from/to, API and library calls, basic block information, flags, arguments, and stack frame information. Also, the features are grouped together by their categories labeled as types *t*. Thirdly, region vectors $\overrightarrow{V}_{B_R}$ are also generated by marking the regions in assembly, which contain a distinctive property according to an entropy-based analysis [20]. For instance, if the ratio of arithmetic/logical operations is higher than a predefined threshold, the region is marked as potential crypto operations.

Likewise, a flag is set for identified packed code according to PE section sizes by comparing

the ratio of virtual size to the size of the raw data. Fourthly, for detecting APIs and system calls, a recursive traversal through wrapper functions might be necessary to spot the target call. This is achieved by identifying the potential wrappers and storing only the final API as a feature. There could be correlational patterns between some features in the assembly vector. For instance, Figure 14 shows the control flow graph (CFG) of a function, which calls an AES encryption function from the OpenSSL library [36] to encrypt data before transmitting it over a socket connection. In this case, prior to the encryption operations, a call is made to set the buffer memory along with the encryption key. An analogous scenario holds true for the decryption case. Therefore, an emergent pattern can be observed, which includes similar values for vector elements. In this case, a frequent sequential pattern is formed that can be used to identify calls to crypto-related functions. Finding such patterns are important in classification steps for recognizing general classes of code functionality.

4.2.7 Function Classification Based on Assembly Vectors

Following the generation of $\overrightarrow{V}_{B_f}$ for all functions in assembly form, a classification can be made according to the similar features which could be an indication of potential code functionality. Figure 15 is an example of feature extraction from a piece of assembly. Initially, a dataset *ds* is built using the set of $\overrightarrow{V}_{B_f}$ binary vectors and the features are then refined using a normalization phase. This process is used for validating feature values and it filters out illegal characters according to a predefined blacklist. The result is a normalized version of the dataset *ds*. Besides, this function generalizes certain features of the assembly associated with memory addresses, registers and data values. At this point, the dataset can be queried based on user-specified attributes.

A matrix structure is created by function $B \in ds \rightarrow [\mathrm{m}_{ij}]^{v_n}$ based on a particular feature v_n or for all selected features. For instance, in the following data matrix, A_i indicates the API call labels and f_i represents functions in the disassembly. The letters a, b, c, d, ..., indicate the number of occurrences of each API call. The numbers are used for term frequency analysis. Depending on the analysis scenario, a different set of features can also be used in place of each A_i.

$$[m_{ij}]^{A_n} = \begin{bmatrix} & A_1 & \cdots & A_n \\ f_1 & a & \cdots & b \\ \vdots & \vdots & \ddots & \vdots \\ f_m & c & \cdots & d \end{bmatrix}$$

A similar structure with binary values is also used for bitmap indexing for a quick search in vectors. The bitwise structure is beneficial for dimension reduction and for replacing string values with single bits, which leads to computational savings. A similar matrix can be formed for other features with the purpose of function classification and for grouping the functions based on similar classes of API calls.

The idea is to learn the mappings between the representative feature vectors of source, namely $\overrightarrow{V}_{\rho\psi}$ and assembly level vector $\overrightarrow{V}_{B_f}$ for the files in each semantically consistent path ρ_ψ. This will help for grouping the representative functions into the right clusters according to a higher level semantic functionality. Despite the fact that the exact lexical elements of a program Prg_i can be re-arranged to form a new program, the inherent information about the program's domain, which is expressed through the path ρ_ψ helps classify the program into the correct group of functionality.

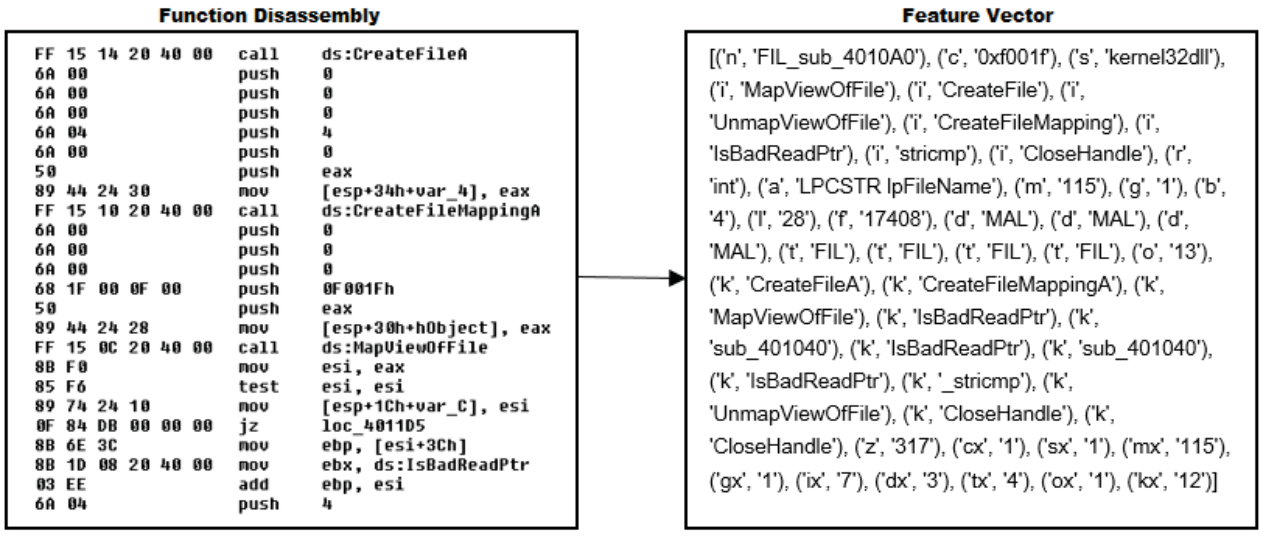


Figure 15: Assembly Feature Extraction.

For instance, it can be seen that for many crypto-related functions, certain features such as I/O operations, file stream manipulation, buffer flushing, calls to encryption or decryption functions, and permission attributes are in common, regardless of the way the program is organized.

4.2.8 Matching Binary-to-Source Patterns

A data matching process involves several key steps [18]. The primary step is to refine the attribute and value pairs of two vectors to ensure integrity and consistency. Due to the high computational complexity of data matching, an indexing step is required for efficient data handling. The index is then used in a pairwise comparison step for finding the potential matches. The final outcome of this process is a tag from the result set of $Res = \{M, N, P, Z\}$ where the elements stand for match, non-match, potential-match and no-decision respectively.

1) **Mapping Rules and Search Based on Language Structures**

Assuming that the language L was used to create Prg_i under the development environment $\mathcal{D}ev$, a

set of *mapping rules* M_i is predefined for describing the relationship between the multiset of structures in program's assembly and structures associated with keywords ω_i from the set of language keywords $\sum_L = \{\omega_1, \ldots, \omega_n\}$. For instance, for a binary program Prg_c written in C++, the rules M can help correlate assembly-level to source-level features such as the primitive and composite data-types, structures, arrays, methods, constructors, destructors, virtual and non-virtual functions, and other language specific features such as name mangling, method overloading, inheritance, etc. Once the mapping rules M_i have been defined, a search in $\mathcal{S}$ can be done by supplying pairs in the form of *term*:*structure* similarly to the online analysis [47]. This searching scheme is useful for finding source-level features such as class methods, constructors, destructors, attributes, type declarations, interfaces, objects and fields.

2) **Incremental / Active Learning Process**

The process of incremental learning is initiated using the normalized *ds* dataset, which contains the instances of reference base classes. Each sample in this dataset specifies a class tag *C* and the joint feature vectors of assembly and source in form of C:$\{S_1^{src}, B_1^{asm}, \ldots, S_n^{src}, B_n^{asm}\}$ where the values of S_i and B_i (standing for source and binary feature vectors) are obtained from generalization of features in ρ_ψ, $\overrightarrow{V}_B$ according to the mapping rules M_i. The initial classifier training is completed with reference patterns in *ds*. For previously unseen vectors with classification results of P (potential match) or Z (no-decision), a manual review is performed and the corresponding tags and feature vectors are used as active learning samples for improving the accuracy of the supervised learning process. There are several learning models that are suitable in this scenario.

3) **Multi-Class Support Vector Machine Classifer**

Algorithm 1: Feature Extraction and Training for Reference Classes

```
/* (I) Building a Customized Local Code Repository                    */
   Input  : P = {ρ^1, ..., ρ^k}, T_n = {Γ_1, ..., Γ_n}, SF
   Output: local_repository with tagged files, i.e., f_L<i, ρ, τ>
RepositoryBuilding =
    foreach ρ in P do
        foreach f_L in path ρ do
            assign (f_L, τ, i);
end
/* (II) Generating Programs                                           */
   Input  : Dev, CL, Src ⊆ SF
   Output: CL_Dev, DC, sig set of known signatures
ProgramGeneration =
    Prg, CL_Dev, ASrc, DC := ∅, ∅, ∅, ∅;
    foreach src ∈ Src do
        prg ← compile (src);
        Prg, CL_Dev := Prg ∪ {prg}, CL_Dev ∪ {src ↦ prg};
    foreach prg ∈ Prg do
        asrc ← disassemble (prg);
        ASrc, DC := ASrc ∪ {asrc}, DC ∪ {prg ↦ asrc};
    foreach asrc ∈ ASrc do
        V⃗_{B_F}, V⃗_{B_R} ← extractFeatures (asrc);
        B_{sig_n} ← matchSignature (asrc, sig);
end
/* (III) Extracting Source Level Features                             */
   Input  : ls, parsing mechanism T
   Output: Par, Sig, λ
SrcLevelFeatureExtraction =
    Par, Sig, λ := ∅, ∅, ∅;
    foreach ρ_ψ do
        foreach file in ls[{ρ_ψ}] do
            proto ← extractPrototype (file);
            sign ← genSignature (proto); //generate signatures
            V⃗_{ρψ} ← applyMechanismT (file, proto, sign);
            // generate representative feature vector
            Par, Sig, λ := Par ∪ {file ↦ proto}, Sig ∪ {proto ↦ sign}, λ ∪
            {file ↦ V⃗_{ρψ}};
end
```

Algorithm 2: Matching Decision for Target Files

```
   /* (I) ASM to Source Matching                                          */
1    Input  : the set of target_file functions, set of M_i mapping rules
2    Output: V_BF, V_BR for the target file and on-line analysis results
3  ASMToSrcMatching =
4      V_BF, V_BR ← extractFeatures (target_file);
5      refine (V_BF); //based on a dictionary of black list characters
6      foreach function in target_file do
7          onLineAnalysis (function, local_repository);
8  end
   /* (II) Classifying each Function of the Target File                   */
9    Input  : the set of target_file functions and Res = {M, N, P, Z}
10   Output: classification tag for each function
12 FunctionClassification =
12     foreach function in target_file do
13         res ← classify (function);
14         if res ∈ {P, Z} then
15             manualReview (function);
16             updateTrainingSet (function);
17             incrementalLearning (res, V_BF, V_BR);
18         displayClassificationResults ();
19 end
```

We utilize Support Vector Machine (SVM) as our classifier in the experiments. SVM is a promising supervised learning model for classification (SVC) and prediction (SVR) that can be applied on linear and non-linear data [28]. It works by transforming the initial dataset into a higher dimensional space in which it searches for one or more hyperplane decision boundaries. During the training phase, the SVM tries to maximize the marginal hyperplane for separating the classes. Several approaches are taken for dealing with a multi-class problem. One solution is to treat the problem as multiple binary classification problems for separating one class against all others. Alternatively, a pairwise classification method is used so that one SVM is employed for each pair of classes. The final classification label is determined by the classifier which scores the highest

value or by a voting heuristic. For an *n*-class problem, $n(n-1)/2$ classifiers are trained in the *one against one* approach and *n* classifiers in the *one against all* scenario.

Support vector classification solves the problem of $min_{w,b,\xi}(\frac{1}{2})w^T w + C\sum_{i=1}^{n}\xi_i$ subject to $y_i(w^T\varphi(x_i)+b) \geq 1-\xi_i$, $\xi_i \geq 0, i = 1,\ldots,n$ where $x_i \in R^p, i = 1,\ldots,n$ is the two-class training vector and $y \in R^n$ is a vector such that $y_i \in \{1,-1\}$. The model is also denoted as $min_\alpha \frac{1}{2}\alpha^T Q\alpha - e^T$ subject to $y^T\alpha = 0, 0 \leq \alpha \leq C, i = 1,\ldots,l$ where *e* is an all one vector, *Q* is an $n \times n$ matrix, $Q_{ij} \equiv K(x_i,x_j)$ and $\varphi(x_i)^T\varphi(x)$ is the kernel. The function φ maps the training vectors into a higher dimensional space. Lastly, the classification decision is made using the $sgn(\sum_{i=1}^{n} y_i\alpha_i K(x_i,x)+\rho)$ function [37].

4.3 Experimental Results

We have conducted two sets of experiments to evaluate the proposed methodology. The first experiment demonstrates the performance of the system in identifying four functionality classes of cryptography, file processing, service manipulation, and networking using only the assembly level features as input. The second scenario demonstrates function classification in malware binaries with regards to the same four base classes.

4.3.1 Classifier Training with Reference Functionality Classes

Malware functionality can be analyzed from different perspectives and the proposed methodology can be implemented based on the scenario. In this experiment, we are interested in identifying the potential malware functionalities that are related to data encryption and decryption, processing of

Class Tag		Number of Attributes					
		Src	**Reduced**	**ASM**	**Reduced**	**Joint**	**Final**
1	Crypto	3674	14	1392	29	43	31
2	File	237	9	1061	24	33	31
3	Service	50	5	974	15	20	21
4	Network	281	5	1158	24	29	21

Table 4: Initial Numbers of Features for the Base Classes.

files, manipulation of system services, and network connectivity. Therefore, we define a set of four functionality class labels of $F = \{C, F, S, N\}$ respectively.

1) Data Pre-Processing and Feature Groups

The training data are processed in several steps according to the methodology before they are used for classifier training. During these steps, the number, structure, types, and values of the attributes are verified and inconsistencies are resolved. Also, during the feature selection process, an effort is made for retaining the most informative attributes. We have used 40 source files for each functionality class from the corresponding path in the repository and generated the reference patterns. The crypto class `C` has been defined based on all functions in the OpenSSL library [36] and crypto-related Windows API functions. Table 4 shows the number of features during different steps of data pre-processing for each class. The initial features included 3,485 functions from OpenSSL and 189 from Windows API crypto and hashing functions. To reduce the number of features, we have applied a feature selection and generalization phase to retain 14 general crypto classes instead of the specific function names before training the classifiers. Introducing this layer of abstraction helped with the categorization of functions according to the algorithms of: (1) Windows crypt (2) Windows cert (3) symmetric ciphers (4) public key cryptography (5) certificates (6) hash functions

(7) data encoding (8) internal functions (9) SSL session (10) SSL connection (11) SSL cypher (12) SSL context (13) SSL method (14) other function. To generate the assembly from a number of programs that were built with crypto components, we used *IDA Pro* [29] and utilized our customized signature file for OpenSSL libraries.

The file class `F` contained features from Windows API for general file operations, registry manipulation and file search. By applying a similar abstraction for function grouping, we identified 9 features for the reduced vector related to file operations of: (1) Creation (2) deletion (3) copy (4) move (5) compression (6) modification (7) search (8) registry key manipulation and (9) replacement. Each functionality class may include several operations. For instance, all operations such as opening, closing, searching, updating and deleting a key in the registry are considered in the registry key manipulation. An analogous approach was taken for categorizing the features of classes `S` and `N` into 5 groups with respect to service status and network connections.

2) Training Phase and Classification Decisions

The SVM classifier was set to use the linear, polynomial, and RBF kernels in two experiments. Due to the nonlinear nature of data values, RBF kernel was more suited for the scenario. We have split the training set into two parts of training and testing and during the first training round, we performed a pairwise strategy in which one classifier was built for every two classes. Then, we implemented the one against all strategy with 10-fold cross validation. The classifiers were then asked to decide on new generalized feature vectors and the classification was generated in form of a binary vector representing the predicted class. At this point, the training model was ready and it had learnt the mapping between assembly-level and source-level features of the reference classes.

3) Classifier Integration with *RE-Source*

We have integrated the learning models with the offline analyzer of the *RE-Source* framework and evaluated its performance in identifying crypto, file, service, and network functionalities. The offline analyzer is equipped with a module for dealing with assembly features such as API names and function signatures [47]. The output of this module is a statement about the potential functionality of the code and it is inserted as function comments in the assembly file. The purpose of integrating the learning models with the offline analyzer is to enhance the code context recognition and to improve the analysis comments. This enhancement is achieved by inserting a class label before the functionality statement for each assembly function. The feature vectors which are extracted by the offline analyzer from assembly are compared against the reference classes which are formed using joint vectors. Thus, there are several unknown attribute values in each vector and the classifier is expected to find the closest class tag. Certain features should be generalized before the assembly level feature vector is created. The reason behind this generalization is that the learning model knows how to map source to assembly features from a higher level of abstraction. For instance, an API function such as `HttpSendRequestA` is not recognized by the model as the network class. Instead, an abstracted value as `HTP` should be replaced before the matching is initiated. Following this step, a prediction can be made on the assembly level vector.

4.3.2 Classification Results and Performance

It is important to recall that the classifiers have been trained with non-malware features. In other words, it might be expected, that the system performance would be better when non-malware files are analyzed. Although this could be true for packed and obfuscated code, as long as the binary files are generated in similar environments and programming languages, the system is able to give

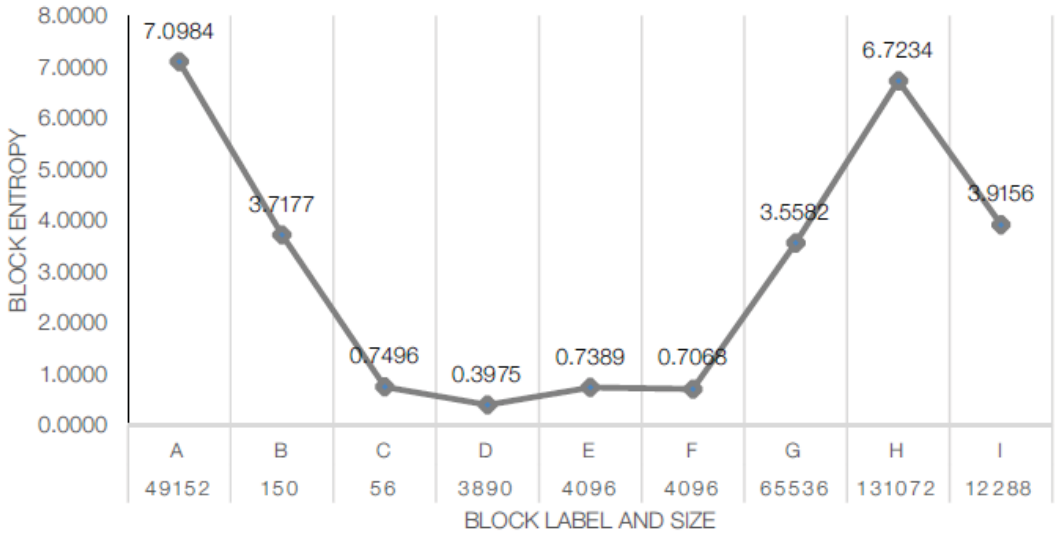


Figure 16: Entropy-based Analysis (first scenario).

meaningful hints about the function semantics. This is due to the generalization steps which had transformed the code into higher units of abstraction. Besides, the learning model had merely learnt the generalized patterns.

In the first part of the experiment, we have tested the offline analyzer on a small set of programs that were formerly used to generate assembly features from the training binaries. In this case, the function class tags were already known and an evaluation could be made on the classification performance. The system precision is calculated as $pre = TP/(TP + FP)$ and the system recall is defined as $rec = TP/(TP + FN)$ for each class. TP, FP and FN stand for true positive, false positive and false negative, respectively. In the first scenario, the precision was 1.0 for the crypto class, 0.88 for file and service classes, and 0.93 for the network class. The overall accuracy was also 0.93 based on 198 classifications. Similarly, the recall of classes were 0.71, 0.98, 0.88 and 0.92 respectively. The highest error rate was associated with network related functions that were tagged as service followed by crypto related functions which were tagged as file class. Figure 16

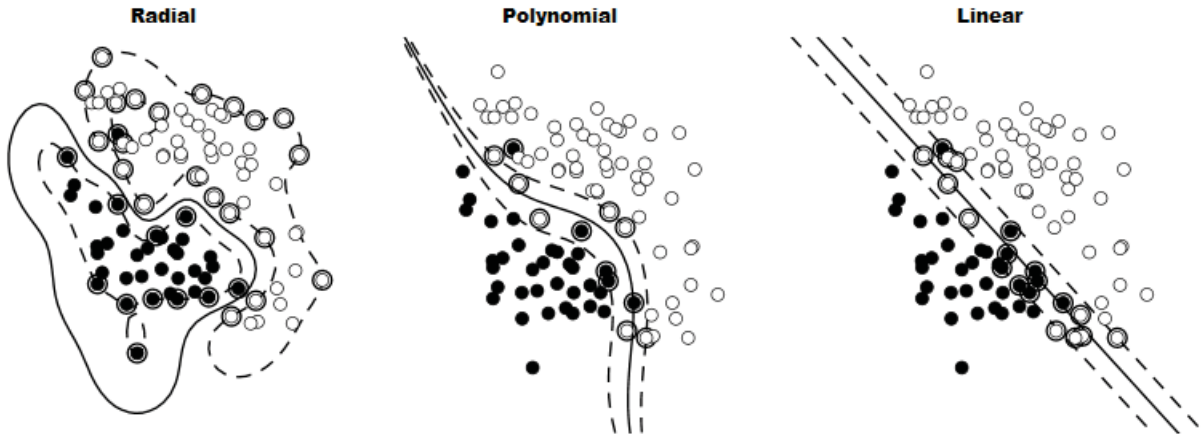


Figure 17: SVM with Pairwise Decision Boundaries for Network and File Classes.

M	Sub	Method	Crypto	File	Service	Net	Total
1	378	API	0	39	11	48	98
		API + SVM	0	45	16	52	113
2	107	API	0	7	0	2	9
		API + SVM	5	11	0	4	20
3	179	API	0	15	2	5	22
		API + SVM	0	17	3	5	25

Table 5: Number of Jointly Identified Functionalities.

shows the entropy of 9 regions of a malware binary. It can be seen that the regions A and H have the highest entropy. Based on a predefined threshold of 6, the graph suggests potential crypto-related functionality in those regions. In this example, the malware is using AES encryption, CRC32 hashing and it embeds an image of another encrypted executable file. Blocks with higher entropy could lead us to encrypted chunks and crypto algorithms. In the second scenario, we have tested the system on three malware samples *M* with (1) trojan (2) ransom and (3) dropper functionalities. The unpacking step was done before testing the malware with the offline analyzer. Based on compiler fingerprinting, the malware binaries were built with VC++ 2008. A combined

decision from the classifier and API module have revealed more about function semantics. In the first sample, the classifiers have unleashed the functionality of 6 more file processing subroutines as well as 5 more service and 4 more network functions. In the ransomware, it has revealed the crypto functionality of 5 OpenSSL and 2 network functions. In the dropper sample, it has helped with the identification of 2 more file and one more service functions. Figure 17 depicts how an SVM model with radial, polynomial and linear kernel partitions the space into decision boundaries. The black and white dots represent data points of significant features in a higher dimension. The marginal data points are associated with functions that have more than one functionality. In the assembly level, these functions may call several APIs from different classes. As it will be discussed in the next chapter, the functions are tagged and renamed using class label prefixes. The prefix `SRV` points to the service class. Similarly, `NET`, `HTP` and `WNT` prefixes show the Windows API networking functions. Likewise, `FIL`, `DIR` and `REG` reference the file class. As can be seen in Table 5, a 10% improvement has been achieved as the result of the learning module. This enhancement is evaluated for the first learning iteration. Over time, the mapping quality will incrementally improve based on the received feedback.

4.4 Summary

We have presented the design methodology for an extension to the *RE-Source* framework. We take an alternative approach for building a semantically consistent local code repository. In this approach, a representative source level feature vector is built that is used as building blocks of a program. The program is then transformed into assembly followed by several normalization

steps. Then, a representative assembly level feature vector is generated that is used for learning the mapping between source and assembly vectors. A classifier is trained to learn the mapping of different program components. Once a new assembly level feature vector is extracted from malware functions, the system is able to reveal the closest functionality class. The methodology has been applied to several malware analysis scenarios. The results show that the approach is able to enhance the matching results even after a single iteration. Over time, the performance will improve by receiving reinforcement feedback from the reverse engineer as active learning. Adopting other multi-class learning models could be the subject of future research extension. Also, the experiment can be done by learning functionalities from malicious code instead of reference patterns. The next chapter introduces a case study on the Citadel malware and confirms the applicability and usefulness of *RE-Source* in real-world malware analysis scenarios.

z

Chapter 5

Case Study: Citadel Malware Analysis

Citadel is an advanced information stealing malware, which conducts targeted attacks against financial information. This malware poses a real threat against the confidentiality and integrity of personal and business data. Recently, a joint operation was performed by FBI and Microsoft Digital Crimes Unit in order to take down Citadel command-and-control servers. The operation caused some disruption in the botnet but has not stopped it completely. Due to the complex structure and advanced anti-reverse engineering techniques, the Citadel malware analysis process is challenging and time-consuming. This allows cyber criminals to carry on with their attacks while the analysis is still in progress. In this chapter, we present the results of the Citadel reverse engineering and provide additional insights into the functionality, inner workings, and open source components of the malware. In order to accelerate the reverse engineering process, we propose a clone-based analysis methodology. Citadel is an offspring of the previously analyzed Zeus malware. Thus, using the former as a reference, we can measure and quantify the similarities and differences of the new variant. Two types of code analysis techniques are provided in the methodology namely

assembly to source code, and binary clone detection. The analysis results prove that the approach is promising in Citadel malware analysis. Besides, the same approach can be applied to other malware analysis scenarios.

This chapter is organized as follows. Section 5.3, is dedicated to explaining our methodology in studying the malware. Section 5.4 details the dynamic analysis of the malware. It starts by revealing the process of infection, followed by explaining the debugging process and memory forensic approaches used for dynamically analyzing the malware. The main features of the Citadel malware is also described in this section. Section 5.5 presents the static analysis of the malware and the steps led to the actual de-obfuscation of the malware binary. Section 5.6 presents our clone-based analysis of the malware where *RE-Source* is proved extremely useful. The threat mitigation is briefly presented in Section 5.7. Finally, the conclusion is given in Section 5.8.

5.1 Introduction

One of the offspring of Zeus which has been making headlines in recent months *(March 2013 - July 2013)* is called Citadel. The cyber criminals behind the Citadel malware have stolen more than 500 million dollars from online bank accounts [55]. Zeus was a prolific information stealing Trojan that has been around since 2007. In 2011, its source code was leaked on the internet and became available to the underground community. Since then, many different malware were generated basing their core on the Zeus code. Similarly, Citadel has been employed by botnet operators to steal banking credentials and personal information. In addition, Citadel has features that extend beyond targeting financial institutions. Spying capabilities such as video capture is an example of

such features that literally enables criminals to collect anything from a victim's machine. The malware also provides ransomware and scareware in attempts to extort money directly from victims. Reverse engineering is often considered as the primary step taken to gain an in-depth understanding of a piece of malware. However, it is a challenging and time-consuming process especially in the case of sophisticated malware such as Citadel which is an evolved variant of Zeus.

The major objectives of this chapter are to reverse engineer the Citadel malware and gain more insights into its structure and functionality using, among other tools, our *RE-Source* framework. In particular, the objectives can be summarized as follows:

1. Quantify the similarity between the Citadel malware and the Zeus malware.
2. Get additional insights into the Citadel open source components.
3. Accelerate the process of Citadel reverse engineering.

To enhance and speed up the process, a new approach is employed in this study, which is called clone-based analysis. Indeed, we illustrate the usefulness of the proposed approach in the analysis of new variants of a malware family.

The main contributions of this chapter are three folds. First, a new methodology for reverse engineering malware is proposed. This methodology significantly decreases malware analysts' efforts and saves time. Second, the similarity between the Citadel malware and the Zeus malware is precisely quantified. Also, additional insights are provided into the open-source components used in the Citadel malware. Third, a detailed reverse engineering analysis of the Citadel malware is presented and its main features are described.

This case study has been chosen for a number of reasons. First, Citadel and Zeus are the real threats against confidentiality, integrity and availability of information systems. Cyber criminals

are constantly enhancing their tools for gaining access to personal and financial data. The profitability of such crimeware tools in the underground market depends on the timeliness and support for new vulnerabilities. Therefore, malicious developers often reuse all or parts of an existing piece of software component during their incremental development process in order to save time. This leads to leaving fingerprints of previously analyzed malcode on the new releases. Clone-based analysis comes in handy in such situations due to its potential for producing quick results. Integrating a clone-based analysis in the reverse engineering cycles could significantly reduce the overall analysis time. The second benefit of this case study is that it allows us to leverage our developed tools such as *RE-Source* [47] and *RE-Clone* [49] in reverse engineering sophisticated malware and gain invaluable insights into ways of progressing them further. The lessons learned during the analysis would bring new opportunities for the future extensions of our tools developed in our lab. Third, the analysis provides us with practical solutions for mitigating future threats in a timely manner. Once the analysis is performed on Zeus and Citadel, any new malware with shared components can be analyzed quickly. Besides, this process will be automated and it provides useful information to the reverse engineer in an effort to reduce the manual work.

5.2 Clone-based Analysis

Clone-based malware analysis can be applied for complementing the process of reverse engineering. Particularly, it could be helpful in reducing required time for the static analysis phase. In this context, two techniques are taken into account for quantifying the similarities between Citadel and Zeus samples. The first approach focuses on assembly to source code matching i.e., applying

RE-Source in order to reveal the open-source building blocks of the malware. Likewise, a second approach is utilized for performing the assembly to assembly code matching.

5.3 Methodology

As already mentioned in Chapter 2, static analysis and dynamic analysis are the two main approaches used in studying malware [41, 45]. Static analysis describes the process of analyzing the code of a malware to determine its structure and functionality without executing it. In contrast, dynamic analysis is the process of monitoring the malware behaviors on a victim machine as it is executed or after the machine is infected. In general, the process of reverse engineering malware is a combination of these two approaches which is time-consuming and costly. The success of these approaches are tightly coupled with the functionalities of the tools and skills of the reverse engineer [43, 44].

To enhance and accelerate the process in analyzing the Citadel malware another dimension is considered in our study as shown in Figure 1. This new dimension is called clone-based analysis. In few words, the clone-based analysis identifies the pieces of code in Citadel malware that are originated from other malware and open-source applications. This step is performed automatically by leveraging the tools that are designed and developed in our security lab [47, 49]. There are two main advantages in considering this extra dimension into the static analysis. First, to avoid dealing with low-level assembly code in situations when its corresponding high-level code is available. Second, to prevent reverse engineering parts of the malware which has already been analyzed. This approach is very promising, especially in the cases similar to Citadel which shares a significant

portions of code with a previously reverse engineered malware like Zeus [46]. The process of assembly to source code matching is performed using the *RE-Source* framework. Also, the binary clone matching is done using *RE-Clone*.

Three concurrent processes are defined in the proposed methodology. The dynamic analysis track focuses on web debugging, memory forensics, process injection and web injects. An important aspect in this process is the observation of malware's behavior in response to controlled inputs.

On the other hand, the static analysis process focuses on assembly-level functions. De-obfuscation could occur in the overlapping area of these two methods. Unpacking and decryption are relevant examples that fall in this area. It is assumed that a database of previously analyzed code is available during the analysis. Code search engines provide an interface to online open source code repositories. Likewise, an offline code repository is maintained for storing the malware assembly code and the results of previous analysis sessions. One advantage of the clone-based analysis is that it could guide the dynamic and static steps. In other words, it highlights the important directions that the other two processes should follow by eliminating the clones, distinguishing the library functions, and providing additional comments. Therefore, the analysis focus is shifted to the different parts of the new malware, resulting in a shorter analysis timeline.

Citadel and Zeus droppers binaries can be customized using the bot builder or third-party tools. Therefore, there might exist several MD5 hashes which refer to the same dropper family. In this analysis, two samples of Zeus and five samples of Citadel have been used. In the following sections, the main steps and results of the malware analysis are presented.

5.4 Dynamic Analysis

The purpose of dynamic analysis process is to execute the malware and monitor its behavior in a controlled environment. Many tools and techniques are available for debugging malware [42, 43]. Sandboxing is a common technique in dynamic analysis and it is used for running untrusted code in a virtual setting. However, modern malware are well-equipped with anti-virtual machine protection against popular tools such as *Oracle VirtualBox* and *VMWare Workstation*. The malware can easily sense whether it is running on a virtual machine by checking certain artifacts in memory or on disk. As a result, the malware might change its normal behavior by taking an alternative execution path for hindering the analysis. Malware can even go one step further and try to exploit the virtual machine vulnerabilities in order to gain access to the host operating system. Thus, successful dynamic analysis may require caution and pre-processing steps. Debugging Citadel is challenging due to the built-in anti-debugging and injection capabilities but they can be overcome by choosing the right strategy. As it will be discussed in Section 5.6, *RE-Source* can provide informative tags such as `ADB`, `PSJ` or `AVM` for functions that potentially contain anti-debugging, process injection, or anti-virtual machine functionality.

5.4.1 The Infection Process

The Citadel bot operates in several modes. Upon the first execution, the Citadel dropper is in the *installation* mode. First, it unpacks and decrypts itself into the memory. Then, it creates a copy of the binary file and stores it in the `%AppData%` folder under a randomly generated sub-folder and file name. The bot file is referred to as *Random.exe* in this context. As an example, the output

Figure 18: Citadel Process Injection and Agent Mode

path could be similar to: ...*AppData**Roaming**Random**Random.exe*. The bot also generates a batch file for removing the installation code. Checking for the existence of this path is a way of knowing whether a system has been infected by the malware. Once the *Random.exe* is run from the new location, a sequence of similar steps are taken for unpacking and decrypting the bot. Afterward, the bot switches to the injection mode and injects itself into the Explorer process and its child processes. The injection step is dependent upon the privileges of the user who runs the bot and the version of the operating system. Following the injection, the bot process is terminated and the installation files are removed. Also, the bot updates the registry and adds an entity so that it will execute each time the operating system boots up. The registry path would appear as: *HKU*\..*Software**Microsoft**Windows**CurrentVersion**Run**Random*. The *Random.exe* is almost identical to the dropper except for the flag bytes located at the end of the file. This portion is encrypted and is used for controlling the bot mode. Therefore, even though the two executables are very similar, their behavior is totally different as the *Random.exe* operates in *agent* and *injection* modes only. On each system startup, the bot performs the injection and initiates the intelligence gathering process as it has been demonstrated in Figure 18.

Figure 19: Decoded Citadel config File Name and Location

5.4.2 Debugging and Memory Forensics

After setting up the analysis environment and infecting it with the malware, the bot execution can be monitored and controlled using a scriptable debugger [58, 59]. Several techniques are available for hiding the debugging process from the bot and gaining more control over the debugger [42]. Besides, a web debugger or a network protocol analyzer is used for monitoring the HTTP network communications of the malware. Citadel encrypts the command-and-control (C&C) network traffic with RC4. Therefore, the crypto keys are required to decrypt the traffic and view the stolen data.

One way of finding the keys is through debugging and setting hardware breakpoints on functions which precede network communication. As it will be discussed in Section 5.6, these functions can be identified through the `NET`, `WNT` and `CRY` tags assigned by offline analysis in *RE-Source*.

Upon successful installation, the bot checks for Internet connectivity and tries to connect to embedded C&C addresses in order to announce its availability. The bot sends requests such as

`POST /carfca/basket.php HTTP/1.1` or `POST /carfca/file .php HTTP/1.1` to the server. The server then replies and sends the encrypted config file to the bot. One major difference between Zeus and Citadel is in the way they handle the transmission of the configuration file. It was possible to find the location of Zeus config file and download it with minimal effort. Whereas in Citadel, it is more difficult to obtain the config file during the analysis. Citadel uses dynamic APIs and it decrypts strings in memory during the execution. This can be considered as an additional layer of protection for the bot internals. Besides, it prevents the config file from being detected easily. Figure 19 shows one of the decrypted links to a Citadel C&C server which hosts the encrypted "*sport.doc*" config file. During the debug, the bot allocates memory for new segments and overwrites the memory space with decrypted code and data from the code segment. The zero values in Figure 19 show the bytes that are yet to be overwritten by data. Blocking the malware's access to the requested C&C and modifying its timing mechanism, will force the malware to enumerate the list of other embedded C&C servers.

Several tools and plug-ins are available for dumping memory, reconstructing import tables, and fixing PE headers. *OllyDump* and *ImpRec* are examples of such tools for unpacking Citadel [41, 43]. *Volatility* [63] was the most versatile and straightforward tool for memory forensics that was used in this project. It automatically builds the import table and generates the executable versions of the unpacked binary. *Volatility* was utilized for creating executable process dumps, retrieving decrypted strings from the memory, and for complementing the static analysis. (See Section 5.5.) Also, it should be mentioned that due to the anti-debugging techniques used in Citadel, the debugging process was one of the complex steps in our study. To accomplish this step, ESET R&D Center, Canada kindly collaborated on the de-obfuscation.

5.4.3 Citadel Attack Configuration

The configuration file is where the bot options are set. This file contains two sections for static and dynamic configurations as depicted in Figure 20. The bot builder tool reads this file and embeds the settings in the generated *bot.exe* file. The bot encryption key is also defined in this file. The static config section is where the options for the initial attack are set. Similarly, the settings for web injects are defined in the dynamic config section. Web injects are used for tricking the users into revealing confidential information such as additional passwords or PINs. Since the man-in-the-middle attack (DLL hooking) occurs in the low-level network libraries such as *wininet.dll* or *nspr4.dll*, the victim user might not be able to distinguish the injected data from the genuine page. The result of injection could be in forms of extra fields, text boxes or warning messages.

In comparison to Zeus, Citadel has a few extra features such as the anti-virus and security software evasion mechanism. Also, the DNS filter enables the bot to block the victim from accessing security-related websites and downloading new updates and patches. Consequently, it makes the machine more vulnerable to future attacks. A DNS redirection technique is used for implementing this feature. Besides, the config file includes a list of blocked websites and the corresponding redirected IP addresses. The report in [57] provides a lists of Citadel DNS filter domains. This type of DNS poisoning attack does not modify the Windows *Hosts* file. It places the network API hooks at a lower level. The settings related to the dynamic configuration can be updated by the C&C server according to predefined rules set by the botmaster. For instance, new modules can be remotely installed for country-specific web injects which target online banking accounts, automatic money transfer, and ransom [53, 57]. The encrypted configuration file can be obtained by capturing the bot traffic and replaying a crafted request in debugging.

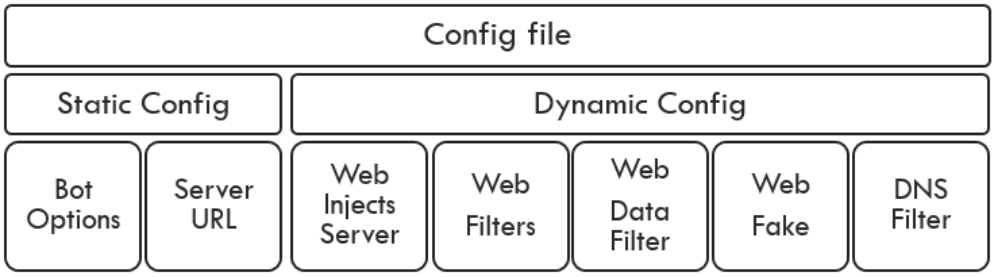


Figure 20: Structure of Citadel Configuration File

In Section 5.5, the process of config file decryption is discussed.

5.5 Static Analysis

In this section, we describe the main steps of the static analysis of the Citadel malware. The static malware analysis process normally starts by disassembling the malware binary. However, the initial disassembled code may not draw a complete picture of the original code due to different layers of obfuscation. Disassembling the Citadel malware using *IDA Pro* [59] results in a packed binary containing merely 13 functions, 11 imports, and 337 strings. The binary was compressed, encrypted, and employed anti-reverse engineering techniques. Therefore, our static analysis started by de-obfuscating the malware. The steps are described in detail in the following sub-sections.

5.5.1 Unpacking Step

Not surprisingly, the malware was packed with a non-standard packing scheme. Therefore, automatic unpacking tools such as *UPX* could not be used for unpacking the binary and manual unpacking was required. To unpack the malware, a combination of static and dynamic techniques was used. The packed binary was executed in *Immunity debugger* [58] until the unpacking stub

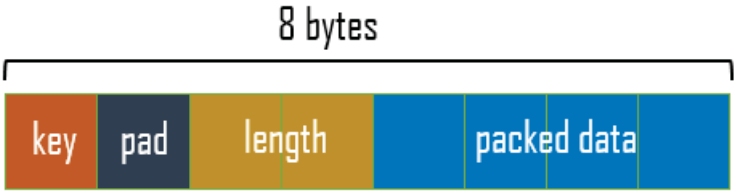


Figure 21: Structure of the Encrypted Data

unpacked the binary in memory. Once the unpacking procedure was completed, the unpacking stub transferred the execution to the original entry point of the binary by making a jump from one segment to another segment. At this moment, *Volatility* [63] was used to dump the unpacked version of the binary's process out of memory and generate an executable unpacked version of the binary. The generated binary contained about 800 functions, 386 imports, and more than 900 decrypted strings and enabled the static analysis to be progressed.

5.5.2 Code Decryption Step

After unpacking, there were still some encrypted portions in the binary code. One of the interesting portions was located in the address of `0x0040336` in our sample. In-depth examination of the function which cross-referenced this portion, revealed the structure of the encrypted data and its decryption algorithm. As shown in Figure 21, the size of the structure is 8 bytes and consists of 4 chunks. Also, the key for string decryption is embedded in the binary file.

Algorithm 3, presents the decryption procedure used for decrypting the data. By writing the Python script, more than 300 strings and 45 C&C commands were retrieved statically from the binary code.

Algorithm 3: String Decryption Algorithm

```
/* Python command for decrypting the embedded strings */
for j in range length do
    UNPACKED_DATA = join(char(PACKED_DATA[j]) ^ j ^ key)
```

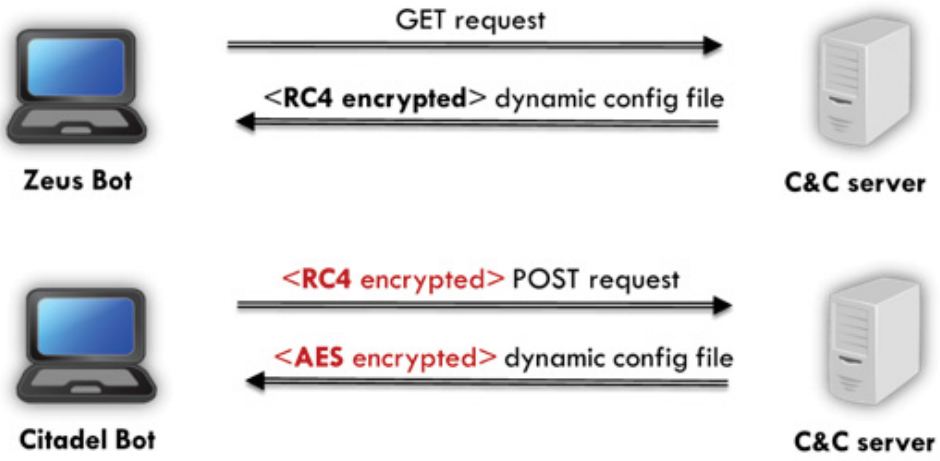


Figure 22: Communication Messages for Retrieving the Configuration File

5.5.3 Crypto Algorithms

Receiving an RC4 encrypted configuration file from a C&C server in response to a plain `GET` request was a weaknesses of the Zeus malware. To overcome this, significant improvements have been taken place concerning crypto algorithms in the Citadel malware. As shown in Figure 22, unlike Zeus, the Citadel C&C server expects a specially crafted RC4 encrypted `POST` message to return the configuration file. In addition, in order to provide a better security, the configuration file is encrypted using AES CBC128.

The Citadel malware authors have used a composition of different ciphers. Figures 23 and 24 present the details of the crypto algorithms used in the Citadel. The RC4 encryption (Figure 23) starts by a customized XOR operation, which is called Visual Encryption. In fact, this algorithm is

Algorithm 4: Visual Encrypt Algorithm

```
1 void Crypt::VisualEncrypt(void *buffer, DWORD size) {
2 for (DWORD i = 1; i < size; i++) do
3
4     ((LPBYTE)buffer)[i] ^ = ((LPBYTE)buffer)[i − 1];
5 }
```

a form of encoding/obfuscation. The input to the algorithm is an encoded buffer. The code of the Visual Encryption is provided in Algorithm 4. This function was used in Zeus for crypto purposes as well. After the XOR operation, the non-standard RC4 initialization routine generates a 0x100 bytes key based on the static configuration data embedded in the binary. The output of the routine is a new RC4 key that is used in RC4 encryption function along with the customized XOR-ed data. Finally, performing an XOR on the RC4 output and the login key embedded in the binary, results in the RC4 encrypted data. Given *login_key*=*lkey* and *Visual_Enc*=*encode*, the functionality can be describes as: *out* = *lkey* XOR $RC4_{rkey}$*(encode(in))*. Therefore, *out*=Enc(*in*).

Figure 24, describes the AES decryption of the Citadel malware. The Configuration Decryption routine, takes the static configuration data embedded in the binary as input, and outputs the embedded RC4 key. Additionally, the login key is hashed using MD5 method. Afterward, the hashed login key and the embedded RC4 key is fed to the RC4 routine. Next, performing an XOR on the output of the RC4 routine with the login key results in the AES key. This key is fed to the AES decryption method. As the last step, the Visual Decryption function takes the result of the AES routine and generates the decrypted data. The pseudo-code of the Visual Decryption is provided in Algorithm 5. The process can be formulated as: AES_{key} = H(*lkey*) XOR $RC4_{rkey}$. Given *Visual_Decrypt*=*decode*, the output can be stated as: *out* = *decode(*AES_{AES_key}(*in*)). The weakest point in the whole encryption/decryption process is that it is based on static configuration data.

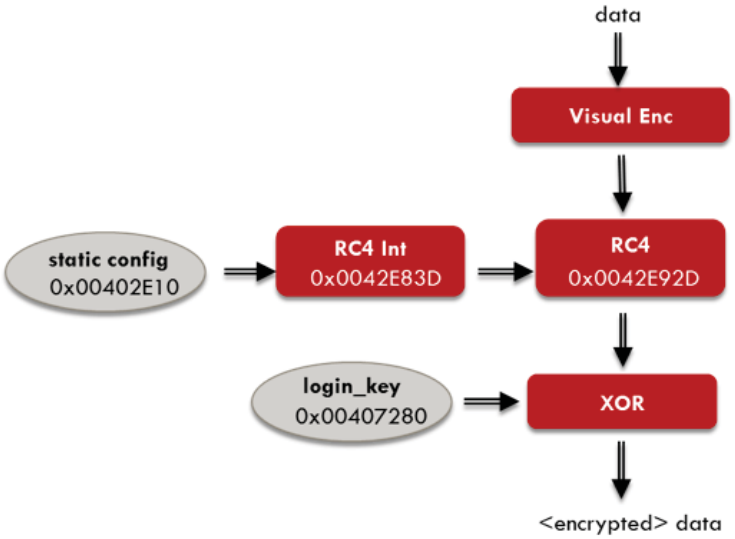


Figure 23: Citadel RC4 Encryption Process

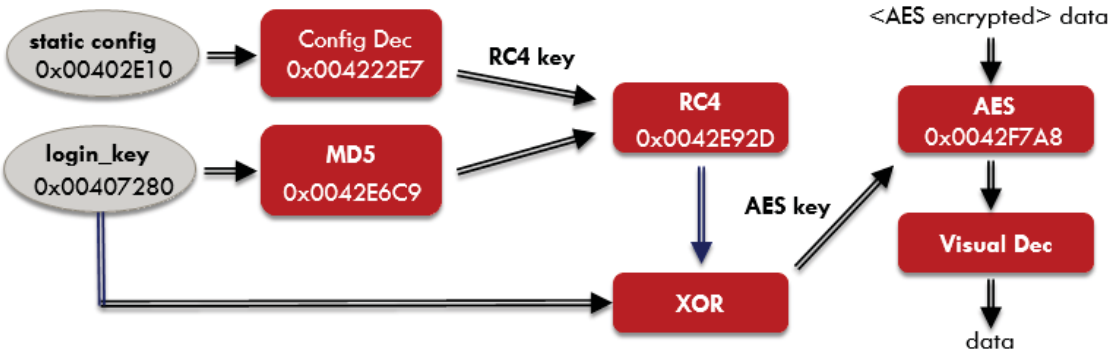


Figure 24: Citadel AES Decryption Process

Also, it demonstrates lack of competency in security algorithms by the malware authors.

5.6 Clone-based Analysis

Clone-based malware analysis can be applied for complementing the process of reverse engineering. Particularly, it could be helpful in reducing the required time for the static analysis phase. In this context, two techniques are taken into account for quantifying the similarities between Citadel and Zeus samples. The first approach uses *RE-Source* in order to reveal the open-source building

Algorithm 5: Visual Decryption Algorithm

```
1 void Crypt::VisualDecrypt(void *buffer, DWORD size) {
2 if size > 0 then
3     for (DWORD i = size − 1; i > 0; i−−) do
4
5         ((LPBYTE)buffer)[i] ^= ((LPBYTE)buffer)[i − 1];
6 }
```

blocks of the malware. Likewise, a second approach is utilized for performing the assembly to assembly code matching. The major steps in the clone-based methodology can be enumerated as follows: (1) Identification of standard algorithms and open source library code in the malware disassembly, (2) assigning meaningful labels to assembly-level functions based on API classification, (3) commenting the assembly code based on a predefined dictionary of malware functions, (4) applying a window-based search and comparison mechanism for finding the pre-analyzed code components.

5.6.1 Assembly to Open-Source Code Matching

The *RE-Source* framework [47] was used for extracting assembly-level features from Citadel functions. During the online analysis phase, *RE-Source* revealed the correlation between function-level features of Citadel and several open-source projects. As can be seen in Figure 25, the video capture capability of the malware was unleashed through the links to source files such as: *MHRecordContol.h*, *stopRecord.c*, *trackerRecorder.h*, *signalRecorder.h*, *waitRecord.c*, etc. This observation was further supported by occurrences of strings such as *"_startRecord16"* during the dynamic API deobfuscation. Moreover, a "`video_start`" C&C command was also encountered in this process. Even though screen capture is quite common in malware samples, live video capture capabilities

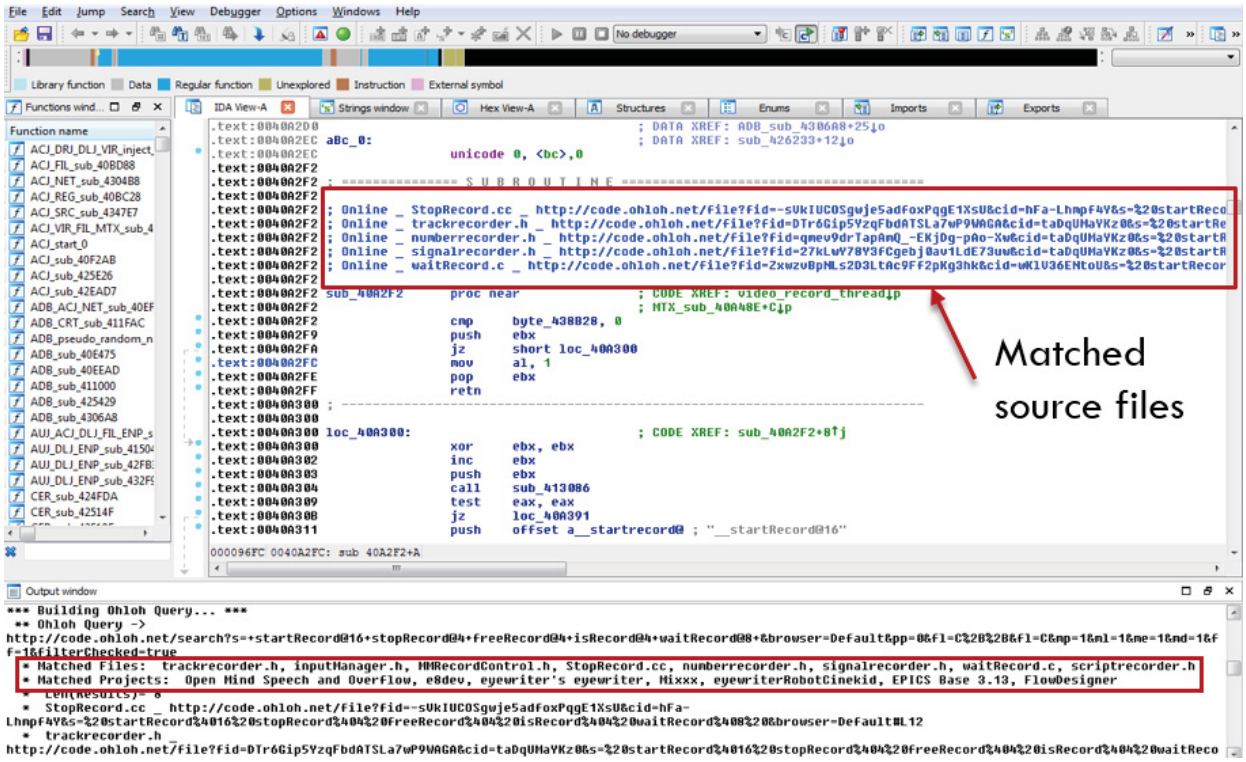


Figure 25: Output of RE-Source Pointing to Video Capture Source Code

are mostly seen in progressive samples.

It should be noted that the online analysis results of *RE-Source* which suggested video-related capability were the outcome of an approximate code matching process. Even though the matching process was not perfect, it was accurate enough to reveal the functionality context in this case. Similarly, *RE-Source* had commented the code with references to other open source projects such as the ones listed in Figure 26. The number of matched projects in each category determines the size of the pie slice in the chart.

Many Zeus-based malware variants have appeared online since the Zeus source code was released in 2011. Having access to Zeus source code enabled us to match Citadel binary against Zeus source code. The pie chart in Figure 26 shows the general categories of open-source projects that

are used in Citadel. Apart from the detached slices, Citadel and Zeus share a considerable amount of code related to the core, VNC, crypto, proxy and math functionality. However, the difference can be summarized in network communication code, new exploits and browser-specific code for web injects.

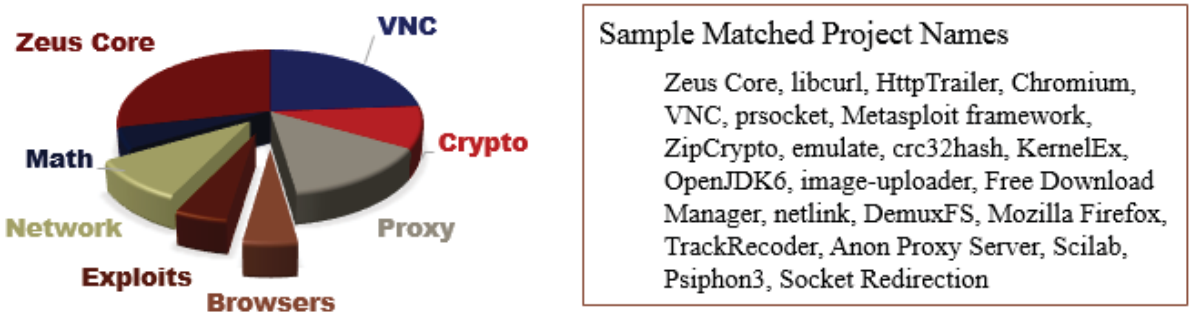


Figure 26: Matched Features with Open-Source Projects

5.6.2 Offline Analysis and Functionality Tags

RE-Source can also be used for tagging assembly functions based on API calls and classifying functions based on their potential functionality. When applied to the unpacked version of Citadel, 652 functionality tags were detected by the offline analyzer. A function is assigned several tags if it contains more than one system call. Accurate functionality tags could convey meaningful hints to the reverse engineer during the static analysis phase. In conjunction with the code and data cross-referencing, functionality tags can enrich the disassembly by highlighting the final system calls in a multi-level function call hierarchy. Since system calls serve as interaction points with the operating system, having a high-level view of them could draw a more organized view of the code.

Functionality tags are not limited to simple system calls merely for file processing or registry

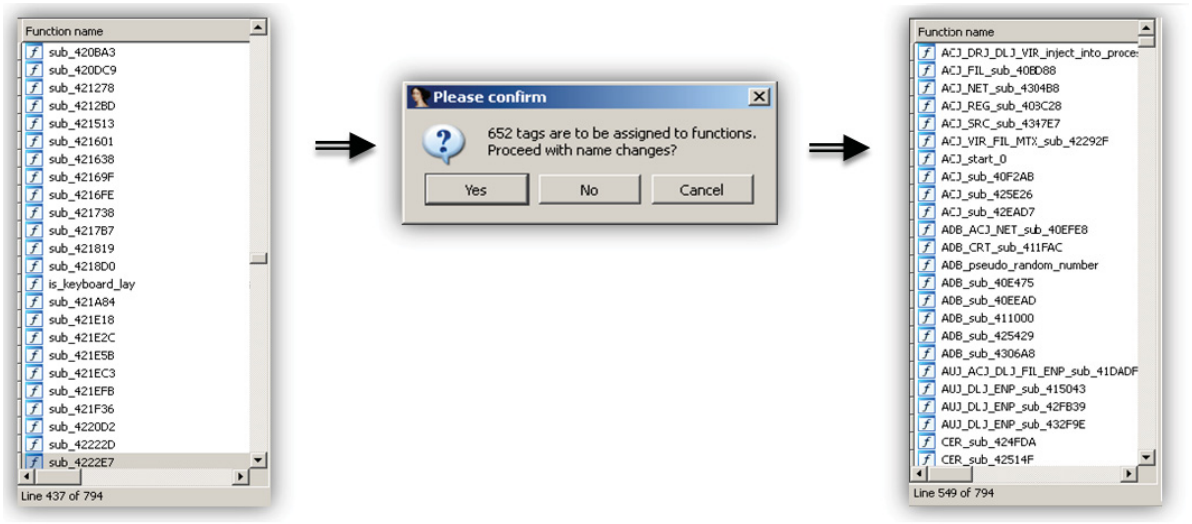


Figure 27: Prefixing the Functions with Functionality Tags

TAG	*Functionality*	*TAG*	*Functionality*	*TAG*	*Functionality*
ADB	Anti-debugging	REG	Registry update	CER	Certificates
PSJ	Process injection	DIR	Directory	OSI	OS Information
DLJ	DLL injection	MTX	Mutex	SRC	Search
DRJ	Direct injection	PIP	Pipe	VIR	Virtual memory
HKJ	Hook injection	HTP	HTTP, Web	CRT	Critical section
ACJ	APC injection	URL	URI links	MOD	Modification
AUJ	APC userspace	ENP	Enumeration	SRV	Service
AKJ	APC kernerlspace	HAS	Hashing	LCH	Launcher
WNT	Win networking	CRY	Cryptography	AVM	Anti-VM
NET	Low-level socks	FIL	File processing	CSH	Cache

Table 6: Functionality Tags for Offline Analysis

modifications. They can be composed of compound operations related to common malware behavior. Put another way, several patterns can be defined for highlighting common malicious code in downloaders, launchers, reverse shells, remote calls and keyloggers based on the combination of several simple system operations. In this context, process memory modification and code injection points are of great interest in the analysis. For that, *RE-Source* includes category information such as *process injection*, *launcher*, *DLL injection*, *process replacement*, *hook injection*, *APC injection* and *resource segment manipulation*. Table 6 lists the available functionality tags in the current prototype for conducting offline analysis.

An alternative practical usage of functionality tags is in disassembly comparison of two similar malware variants. Instead of comparing the files by address, the code can be analyzed offline and the generated tags can be used as association criteria. In this process, the functions are sorted based on the assigned tags and the ones with similar tags are analyzed side by side. This technique was specifically helpful in synchronizing the disassembly of Citadel versus Zeus.

Figure 28 depicts the detected functionality tags. The pie chart sectors are proportional to the number of assembly functions categorized under the same functionality group. The `NET` tag was assigned to 60 functions related to low-level network socks. Also, 41 functions were tagged with `CRT` for critical section objects for mutual exclusion synchronization. Similarly, 36 `FIL` tags were assigned to file manipulating functions. The other tags such as crypto, hashing, search and code injection were also identified during the analysis. The `CRY` (crypto) and `HSH` (hashing) tags provided an easy way of disassembly synchronization between Citadel and Zeus as the slight differences between the assembly files had no effect on the overall functionality group.

Translated into quantifiable terms, Table 29 shows the output of *RE-Source* for Citadel vs. Zeus

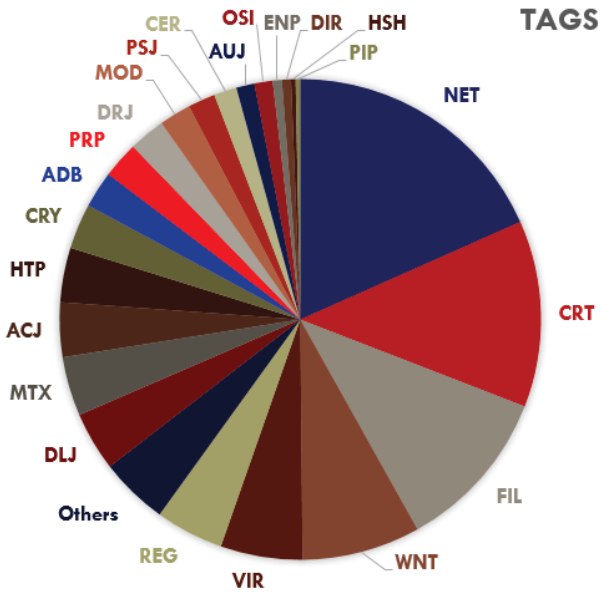


Figure 28: Functionality Tags Assigned by Offline Analysis

	Assembly Functions	Functionality tags	Malware API	Tagged Functions	Source URLs	Matched Projects	Imported Functions	Unicode Strings (3+)
`Citadel 1.3.5.1`	788	652	173	318	293	81	386	917
`Zeus 2.1.0.1`	565	461	149	250	185	56	350	955

Figure 29: RE-Source Analysis Results

Figure 30: Code Analysis After Clone Elimination

comparison. The numbers are reported in accordance with occurrence of certain features such as the number of assembly functions, API and functionality tags, common API in malware, number of matched opens source components, imported function calls, and Unicode strings.

5.6.3 Binary Clone Analysis

The malware analysis process can be accelerated by identifying and removing the previously analyzed code fragments. The aim of binary clone analysis is to compare the assembly file of a new binary sample with a repository of analyzed code. The result of this analysis is the set of matched clones. Two important types of clones are considered in this context, namely exact and inexact clones. Exact clones share the same mnemonics, operands and registers. The only difference is in memory addresses. Whereas inexact clones can be regarded as equal up to certain levels of abstraction. The analysis parameters such as search window size, normalization level and detection algorithm play a significant role on the analysis results. These parameters are set according to each analysis scenario. After marking the detected code fragments as clones, the analysis focus is shifted to non-analyzed and new code segments. For performing this experiment, the *RE-Clone* binary clone detector tool was used [49].

The core components of the Zeus malware has been thoroughly studied in [46]. Also, the source and binary files are available online. Therefore, a new Zeus variant can easily be compared

Malware Bot.exe	*Functions*	*Window Size*	*Exact Clones*	*Inexact Clones*
`Citadel 1.3.5.1`	788	15	526	1876
`Zeus 2.1.0.1`	565			

Table 7: Binary Clone Detection Results

against the existing files in order to measure the similarity and detect the potential exact and inexact clones. This analysis is also applicable to finding the additional functions of the new malware variant. Table 7 shows the results of the binary clone matching. According to the results, the two samples share 526 exact binary clones with a window size of 15 instructions. In other words, almost %93 of Zeus assembly code also appears in Citadel. These clones form approximately %67 of the Citadel binary. This analysis highlights the remaining %33 of the Citadel assemblies to be analyzed. Thus, a significant amount of time is saved by disregarding the clones. *RE-Clone* shows the exact address and location of each clone in the disassembly. Furthermore, the remaining functions could be examined in *RE-Source* before the manual analysis process is begun by the reverse engineer. This approach is depicted in Figure 30. The 1876 inexact clones reported by the tool include multiple combinations of regions which also contain the exact clones.

An interesting example of crypt-related clones is the detection of an inexact clone in the RC4 function which is used for encrypting the C&C network traffic. There are a few extra assembly XOR instructions in the Citadel version of the RC4 function. This clone was found with a threshold of 0.8 and a two-combination inexact clone search method. In this approach, each two-combination of features are considered as a cluster. If more than %80 of regions appear in the same clusters, then they are treated as inexact clones. The red segments in Figure 31 highlight these clones.

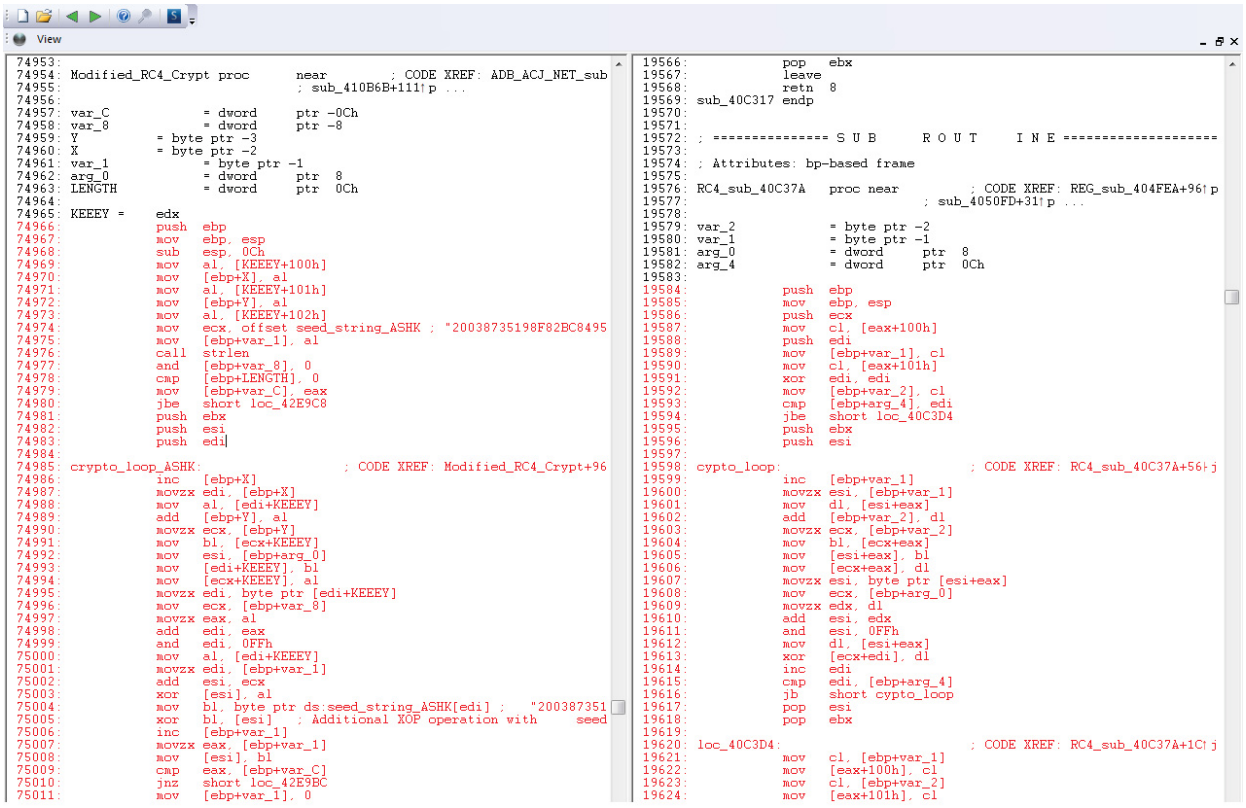


Figure 31: Inexact Clone Detected in RC4 Function. (Citadel vs. Zeus)

5.7 Threat Mitigation by Sinkholing

In June 2013, Microsoft Digital Crimes Unit reported on an operation known as *Operation b54*, in collaboration with FBI to shut down Citadel C&C servers [54]. As a result of this operation, 1400 Citadel botnets around the world were intercepted and redirected to sinkhole servers operated by Microsoft. A comprehensive list of the domain names is available in [56]. Although, the action significantly disrupts Citadel's operation and helps victims from the threat, it has also affected the honeypot systems which were used for identifying and locating the malware creators and distributors. Even though the threat counter-measurement has been successful, cyber criminals can still operate by infecting new machines and operating their bots on alternative servers.

5.8 Summary

The Citadel malware targets confidential data and financial transactions. It is an emergent threat against the online privacy and security. Citadel reverse engineering is challenging as it is equipped with anti-reverse engineering techniques for hindering the malware analysis process. As the number of incidents entailing new malware attacks are increasing, agile approaches are required for obtaining the analysis results in a timely fashion. The malware reverse engineering process consists of two major stages of static and dynamic analysis. This process can be accelerated and enhanced by adding a new dimension for clone analysis. Instead of initiating the process from the scratch, a quick clone-based analysis can easily highlight the similarities and differences between two samples of the same family. The analysis focus is then shifted to the differing sections. We have presented a methodology along with the tools and techniques for analyzing the Citadel malware. Also, we have compared Citadel with its predecessor, Zeus. The similarities have been quantified as the result of two code matching techniques namely assembly to source, and binary code matching. The same methodology can be applied to other malware samples for providing insights into the potential malware functionality. The results of the malware analysis process can be added to a local code repository and used as a reference for measuring the similarities between future samples. They can also be used for improving the accuracy of the results. Overall, the successful completion of our objectives has led to underline best practices for supporting the real-world malware analysis in our laboratory.

Chapter 6

Conclusion

In this research, a new methodology and framework were introduced which can be used for enhancing the process of reverse engineering. Clone-based malware analysis is important for a number of reasons. First, it leads to a significant decrease in the amount of time and effort required during the static analysis process. Put another way, shorter analysis cycles and faster incident responses are the direct outcomes of the proposed approach. Second, the analysis results highlight the potential difference between several variants of malware and provide a means for quantifying the difference. Third, finding the references to source files and open source components of a malware provide valuable insights into the potential functionality and behavior of the malware. Fourth, the functionality tags provide a mechanism for classifying the assembly functions according to API and system calls. Moreover, the tags can provide hints to the reverse engineer for finding the place of specific byte sequences in the disassembly. The important features and aspects of our *RE-Source* framework can be summarized as follows:

- The *RE-Source* framework has been deployed by major organizations for reverse engineering

and malware analysis. It provides a practical, platform-independent, language-independent, scalable, and user-friendly solution for assembly to source matching. It has a graphical interface for setting the search options and analysis configurations.

- The framework provides the support for *online* and *offline* analysis. The online analysis module interacts with the online open source code repositories and the offline analysis module works with a local code repository.

- The output of the online analysis process is a list of project names and file names of the matched source files. It has been observed that a consistency can be achieved amongst the online and offline analysis results if the subject function makes use of standard system libraries.

- The *RE-Source* framework has built-in support for source code parsing, query processing, web scraping, recursive file search, and feature extraction.

- A comprehensive dictionary of common malware functions and their description has been compiled and integrated into the offline analysis module.

- The framework generates datasets based on assembly and source level features. They are saved according to the analysis time and using file and function units.

- The analysis process can be done in automatic and manual modes. The automatic process is repeated for all the functions in the disassembly. In contrast, the manual mode allows the reverse engineer to choose specific features and intervene in the process. Also, different combinations of features can be tried.

- Functionality tags are assigned to function names and code references are added as function comments in the disassembly.

The elaborated framework has been shown effective in reverse engineering and malware analysis. Moreover, the data-mining extension module has been added to the framework for improving the classification results. *RE-Source* was used in several reverse engineering case studies such as the infamous Citadel malware and the results were presented in Chapter 5. In summary, the proposed clone-based methodology introduced a new dimension in reverse engineering and specifically in static analysis.

The limitations of the online analysis are related to the coverage of the local and online code repositories. In other words, the quality of the analysis results depends on the availability of source code with similar features to the assembly. The quality and accuracy of the results can be improved by adding support for new code repositories which host a large number of open source projects. Conversely, the offline analyzer does not solely rely on the source code. It operates based on an incremental learning process and its performance is further improved if it is run on a large collection of disassemblies.

This research can be further expanded in several directions. New support for programming languages other than the C/C++ can be added by replacing the parser with an alternative parsing module. Also, new data mining techniques can be utilized in the offline analysis module. The online analysis module is expandable with heuristic search methods and query processing techniques that take into account domain specific knowledge of the problem and temporal logic for a better accuracy. As a result, the search can be directed to relevant code repositories which leads to better accuracy and precision.

Bibliography

[1] Abrial, J. R.: *The B Book - Assigning Programs to Meanings*. Cambridge University Press, ISBN 052149619-5 (1996).

[2] "Boomerang: A General, Open Source, Retargetable Decompiler of Machine Code Programs," The Boomerang Decompiler Project, 2012. [Online]. Available: http://boomerang.sourceforge.net/.

[3] Bryant, R. E., O'Hallaron, D. R.: *Computer Systems – A programmer's Perspective, 2nd Edition*. Addison Wesley, ISBN 0136108040 (2010).

[4] Eymery, D., Eymery, O., Borello, J-M., Fraygefond, J-M., Bion, P.: *GenDbg : un débogueur générique*. In: *Symposium sur la sécurité des technologies de l'information et des communications SSTIC'08*, France (2008).

[5] "GDB: The GNU Project Debugger," Free Software Foundation, Inc. 2013. [Online]. Available: http://www.gnu.org/software/gdb/documentation/.

[6] "GNU Binutils," Free Software Foundation, Inc. 2009. [Online]. Available: http://www.gnu.org/software/binutils/.

[7] “Google Code: Google’s Site for Developer Tools,” Google, Inc. 2011. [Online]. Available: http://code.google.com/.

[8] “Google Data APIs: A REST-Inspired Technology for Reading, Writing, and Modifying Information on the Web,” Google, Inc. 2012. [Online]. Available: https://developers.google.com/gdata/.

[9] “RE-Goolge: Do you still reverse engineer or do you Google?,” Leder, F., 2009. [Online]. Available: http://regoogle.carnivore.it/.

[10] Lagadec, P.: *Dynamic Malware Analysis for Dummies*. In: *Symposium sur la sécurité des technologies de l’information et des communications SSTIC’08*, France (2008).

[11] “Precise Calculator Project: A Free Scientific Calculator for Windows,” Lastovicka P., 2012. [Online]. Available: http://sourceforge.net/projects/preccalc/.

[12] “Hex-Rays Decompiler: Converts Executable Programs into a Human Readable C-like Pseudocode Text,” Hex-Rays, 2012. [Online]. Available: https://www.hex-rays.com/products/decompiler/.

[13] Troshina, K., Chernov, A., Derevenets, Y.: *C Decompilation: Is It Possible?*. In: *Proceedings of International Workshop on Program Understanding*, pp. 18–27, Altai Mountains, Russia (2009).

[14] Troshina, K., Derevenets, Y., Chernov, A.: *Reconstruction of Composite Types for Decompilation*. In: *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM ’10*, pp. 179–188, Timisoara, Romania (2010).

[15] "Valgrind: A Suite of Tools for Debugging and Profiling," The Valgrind Developers, 2012. [Online]. Available: http://valgrind.org/downloads/.

[16] "Windows Debugging: A Collection of Debuggers and Related Tools for Microsoft Windows," Microsoft Corporation, 2012. [Online]. Available: http://www.windbg.org/.

[17] S. Chaki, C. Cohen and A. Gurfinkel, 2011. Supervised Learning for Provenance-similarity of Binaries. In *ACM SIGKDD Int'l Conference on Knowledge Discovery and Data mining*, New York, USA.

[18] P. Christen, 2012. Data Matching, Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer, ISBN-10: 3642311636.

[19] C. Cifuentes and K. J. Gough, 1995. Decompilation of Binary Programs. In *Software Practice & Experience*, Vol. 25, No. 7, pp. 811–829.

[20] G. Conti, S. Bratus, A. Shubina, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg and R. Perez-Alemany, 2010. Automated Mapping of Large Binary Objects Using Primitive Fragment Type Classification. In *The Proceedings of the Tenth Annual DFRWS Conference*, Vol. 7, pp. S3–S12.

[21] J. R. Cordy and C. K. Roy, 2011. DebCheck: Efficient Checking for Open Source Code Clones in Software Systems. In *19th IEEE Int'l Conference on Program Comprehension*, Washington, DC, USA.

[22] Z. Deng, D. Xu, X. Zhang and X. Jiang, 2012. IntroLib: Efficient and Transparent Library Call Introspection for Malware Forensics. In *Proceedings of the 12th Annual DFRWS Digital Forensics Conference (DFRWS 2012)*, Washington, DC.

[23] R. Elva and G. T. Leavens, 2012. Semantic Clone Detection Using Method IOE-Behavior. In *6th Int'l Workshop on Software Clones (IWSC)*, Zurich.

[24] M. V. Emmerik, 1998. Identifying Library Functions in Executable File Using Patterns. In *Australian Software Engineering Conference*, Adelaide, SA, 1998.

[25] S. Grant, J. R. Cordy and D. B. Skillicorn, 2013. Using Heuristics to Estimate an Appropriate Number of Latent Topics in Source Code Analysis. In *Science of Computer Programming*.

[26] K. Griffin, S. Schneider, X. Hu and T.-C. Chiueh, 2009. Automatic Generation of String Signatures for Malware Detection. In *Proceedings of the 12th Int'l Symposium on Recent Advances in Intrusion Detection (RAID '09)*, Saint-Malo, France.

[27] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I. H. Witten, 2009. The WEKA Data Mining Software: An Update. In *SIGKDD Explorations*, Vol. 11, No. 1, pp. 10–18.

[28] J. Han, M. Kamber and J. Pei, 2012. Data Mining: Concepts and Techniques, Third Edition. Morgan Kaufmann, ISBN-10: 0123814790.

[29] Hex-Rays IDA [Online]. Available from: http://www.hex-rays.com.

[30] E. R. Jacobson, N. Rosenblum and B. P. Miller, 2011. Labeling Library Functions in Stripped Binaries. In *10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools. (PASTE '11)*, New York, USA.

[31] I. Keivanloo, J. Rilling and P. Charland, 2011. Internet-scale Real-time Code Clone Search Via Multi-level Indexing. In *18th Working Conference on Reverse Engineering*, Limerick, Ireland.

[32] C. H. Malin, E. Casey and J. M. Aquilina, 2012. Malware Forensics Field Guide for Windows Systems: Digital Forensics Field Guides. Syngress, ISBN-10: 1597494720.

[33] A. Marcus and J. I. Maletic, 2001. Identification of High-Level Concept Clones in Source Code. In *IEEE Int'l Conference on Automated Software Engineering*, San Diego, USA.

[34] V. Mateljan, V. Juričić and K. Peter, 2011. Analysis of Programming Code Similarity by Using Intermediate Language. In *IEEE Int'l Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia.

[35] A. Ohno and H. Murao, 2006. Measuring Source Code Similarity Using Reference Vectors. In *IEEE Innovative Computing, Information and Control*.

[36] "OpenSSL: The Open Source toolkit for SSL/TLS," The OpenSSL Development Team, 2013. [Online]. Available: http://www.openssl.org.

[37] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher,

M. Perrot, and E Duchesnay, 2011. Scikit-learn: Machine Learning in Python. In *Journal of Machine Learning Research*, Vol. 12, pp. 2825–2830.

[38] I. Santos, F. Brezo, X. Ugarte-Pedrero and P. G. Bringas, 2013. Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection. In *Information Sciences, Data Mining for Information Security*, Vol. 231, pp. 64–82.

[39] S. M. Tabish, M. Z. Shafiq and M. Farooq, 2009. Malware Detection Using Statistical Analysis of Byte-Level File Content. In *ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, New York, USA.

[40] Y. Ye, D. Wang, T. Li, D. Ye and Q. Jiang, 2008. An Intelligent PE-Malware Detection System Based on Association Mining In *Journal of Computer Virology and Hacking Techniques*, Vol. 4, No. 4, pp. 323–334.

[41] M. Sikorski and A. Honig, Practical Malware Analysis, The Hands-On Guide to Dissecting Malicious Software, San Francisco: No Starch Press, 2012.

[42] J. Seitz, Gray Hat Python: Python Programming for Hackers and Reverse Engineers, San Francisco: No Starch Press, 2009.

[43] Malware Forensics Field Guide for Windows Systems: Digital Forensics Field Guides, Waltham: Syngress, 2012.

[44] C. Eagle, The IDA Pro book : The Unofficial Guide to the World's Most Popular Disassembler, San Francisco: No Starch Press, 2011.

[45] A. Singh, Identifying Malicious Code Through Reverse Engineering (Advances in Information Security), New York: Springer, 2009.

[46] H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi and L. Wang, "On the Analysis of the Zeus Botnet Crimeware Toolkit," in *Int'l Conference on Privacy Security and Trust (PST)*, Ottawa, 2010.

[47] A. Rahimian, P. Charland, S. Preda and M. Debbabi, "RESource: A Framework for Online Matching of Assembly with Open Source Code," in *Int'l Conference on Foundations and Practice of Security (FPS)*, Montreal, 2012.

[48] M. R. Farhadi, "Assembly Code Clone Detection for Malware Binaries," MASc. thesis., Concordia University, Montreal, Quebec, Canada, 2013.

[49] P. Charland and B. C. M. Fung and M. R. Farhadi, "Clone Search for Malicious Code Correlation," in *NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)*, Koblenz, 2012.

[50] R. Sherstobitoff, "Inside the World of the Citadel Trojan," McAfee, Santa Clara, 2013.

[51] AnhLab ASEC, "Malware Analysis: Citadel," December 2012. [Online]. Available: http://seifreed.es/docs/Citadel Troja Report_eng.pdf. [Accessed May 2013].

[52] J. Wyke, "The Citadel Crimeware Kit - Under the Microscope," December 2012. [Online]. Available: http://nakedsecurity.sophos.com/2012/12/05/the-citadel-crimeware-kit-under-the-microscope/. [Accessed May 2013].

[53] CERT Polska, "Takedown of the plitfi Citadel botnet," April 2013. [Online]. Available: http://www.cert.pl/PDF/Report_Citadel_plitfi_EN.pdf. [Accessed May 2013].

[54] Microsoft Digital Crimes Unit, "Microsoft, financial services and others join forces to combat massive cybercrime ring," June 2013. [Online]. Available: http://www.microsoft.com/en-us/news/Press/2013/Jun13/06-05DCUPR.aspx. [Accessed June 2013].

[55] J. Vincent, "$500 million botnet Citadel attacked by Microsoft and the FBI: Joint operation identified more than 1000 botnets, but operations continue," June 2013. [Online]. Available: http://www.independent.co.uk/life-style/gadgets-and-tech/news/500-million-botnet-citadel-attacked-by-microsoft-and-the-fbi-8647594.html. [Accessed June 2013].

[56] "List of Domain Names by Registry (Citadel)," June 2013. [Online]. Available: http://botnetlegalnotice.com/citadel/files/Compl_App_A.pdf. [Accessed June 2013].

[57] J. Milletary, "Citadel Trojan Malware Analysis," Dell SecureWorks, 2012.

[58] "Immunity Debugger: The Best of Both Worlds," Immunity, 2013. [Online]. Available: http://www.immunityinc.com/products-immdbg.shtml.

[59] "IDA Pro: Multi-processor Disassembler and Debugger," Hex-Rays, 2013. [Online]. Available: https://www.hex-rays.com/products/ida/debugger/index.shtml.

[60] "Func-Renamer: A Tool for Generically Rename or Comment Functions," A. Hanel, 2013. [Online]. Available: https://bitbucket.org/Alexander_Hanel/func-renamer/.

[61] "BinDiff: A Comparison Tool for Finding Differences and Similarities in Disassembled Code," Zynamics, 2012. [Online]. Available: http://www.zynamics.com/bindiff.html.

[62] “simpliFiRE.IDAscope: An IDA Pro Extension for Easier Reverse Engineering,” D. Plohmann, 2013. [Online]. Available: http://bitbucket.org/daniel_plohmann/simplifire.idascope/.

[63] “The Volatility Framework: Volatile Memory (RAM) Artifact Extraction Utility Framework,” Volatile Systems, 2013. [Online]. Available: https://www.volatilesystems.com/default/volatility.

[64] “Fingerprint: Track a Piece of Malware Based Upon Compile Time, Programming Language Used, Language & Compiler Version,” HBGary, 2012. [Online]. Available: http://www.hbgary.com/community_resources.

Ashkan Rahimian

August 2013