Techniques for the Reverse Engineering of Banking Malware

Paul Black

This thesis is submitted in total fulfilment of the requirement
for the degree of Doctor of Philosophy

School of Engineering, IT, and Physical Sciences
Federation University Australia
PO Box 663
University Drive, Mount Helen
Ballarat, Victoria, Australia 3353

Submitted 16th August 2020

Abstract

Malware attacks are a significant and frequently reported problem, adversely affecting the productivity of organisations and governments worldwide. The well-documented consequences of malware attacks include financial loss, data loss, reputation damage, infrastructure damage, theft of intellectual property, compromise of commercial negotiations, and national security risks. Mitigation activities involve a significant amount of manual analysis. Therefore, there is a need for automated techniques for malware analysis to identify malicious behaviours. Research into automated techniques for malware analysis covers a wide range of activities. This thesis consists of a series of studies: an analysis of banking malware families and their common behaviours, an emulated command and control environment for dynamic malware analysis, a technique to identify similar malware functions, and a technique for the detection of ransomware.

An analysis of the nature of banking malware, its major malware families, behaviours, variants, and inter-relationships are provided in this thesis. In doing this, this research takes a broad view of malware analysis, starting with the implementation of the malicious behaviours through to detailed analysis using machine learning. The broad approach taken in this thesis differs from some other studies that approach malware research in a more abstract sense. A disadvantage of approaching malware research without domain knowledge, is that important methodology questions may not be considered.

Large datasets of historical malware samples are available for countermeasures research. However, due to the age of these samples, the original malware infrastructure is no longer available, often restricting malware operations to initialisation functions only. To address this absence, an emulated command and control environment is provided. This emulated environment provides full control of the malware, enabling the capabilities of the original in-the-wild

operation, while enabling feature extraction for research purposes.

A major focus of this thesis has been the development of a machine learning function similarity method with a novel feature encoding that increases feature strength. This research develops techniques to demonstrate that the machine learning model trained on similarity features from one program can find similar functions in another, unrelated program. This finding can lead to the development of generic similar function classifiers that can be packaged and distributed in reverse engineering tools such as IDA Pro and Ghidra.

Further, this research examines the use of API call features for the identification of ransomware and shows that a failure to consider malware analysis domain knowledge can lead to weaknesses in experimental design. In this case, we show that existing research has difficulty in discriminating between ransomware and benign cryptographic software.

This thesis by publication, has developed techniques to advance the discipline of malware reverse engineering, in order to minimize harm due to cyberattacks on critical infrastructure, government institutions, and industry.

Publications by the Author

[1] P. Black, I. Gondal, and R. Layton, "A Survey of Similarities in Banking Malware Behaviours," *Computers & Security Journal*, vol. 77, Impact factor 3.579, 2018.

[2] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Evolved Similarity Techniques in Malware Analysis," *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE).* IEEE, ERA A, 2019.

[3] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Function Similarity Using Family Context," *Electronics Journal*, Impact factor 2.412, 2020.

[4] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Reanimating Historic Malware Samples", book chapter in *Malware Analysis using Artificial Intelligence and Deep Learning.* Springer, 2020.

[5] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Identifying Cross-Version Function Similarity Using Contextual Features", *IEEE TrustCom*, IEEE, ERA A, 2020 (Accepted).

[6] P. Black, A. Sohail, I. Gondal, J. Kamruzzaman, P. Vamplew, and P. Watters, "API Based Discrimination of Ransomware and Benign Cryptographic Programs", *ICONIP 2020*, ERA A, 2020 (Accepted).

DECLARATION

I, Paul Black, declare that the PhD thesis entitled “Techniques For The Reverse Engineering of Malware” is no more than 100,000 words in length, including quotes and exclusive of tables, figures, appendices, bibliography, references, and footnotes. This thesis contains no material that has been submitted previously, in whole or in part, for the award of any other academic degree or diploma. Except where otherwise indicated, this thesis is my own work.

I give permission for the digital version of my thesis to be made available on the web, via the University’s digital research repository, the Library Search, and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia

Paul Black

15th November 2020

Acknowledgements

I would like to thank Federation University Australia, the Internet Commerce Security Lab (ICSL), and the School of Engineering, IT, and Physical Sciences (SEITPS) for providing me the opportunity to undertake my Doctor of Philosophy in Information Technology.

I would like to express my thanks to Prof. Iqbal Gondal and Associate Prof. Peter Vamplew for guiding me through my research, providing encouragement, answering many questions, and helping me deal with turbulent emotions during my PhD candidature. Special thanks are due to Prof. Arun Lakhotia of the School of Computing and Informatics at the University of Louisiana at Lafayette for agreeing to be co-supervisor.

I would like to thank Westpac Banking Corporation for providing the funding for the ICSL research work that has supported me in the latter part of my part-time Ph.D. The ICSL has given me the privilege of organising the speakers for the Malware and Reverse Engineering (MRE) Conferences for the past eight years. I have enjoyed the challenge of organising speakers for this conference and building contacts with many security researchers.

Table of Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interface.

APTA Augmented Prefix Tree Acceptors.

ASEP Auto-Start Extensibility Points.

BEE Botnet Evaluation Environment.

C2 Command and Control.

CFG Control Flow Graph.

CPU Central Processing Unit.

CVCFS Cross Version Contextual Function Similarity.

DFA Deterministic Finite Automata.

DLL Dynamic Link Library.

DNS Domain Name System.

ECVCFS Enhanced Cross Version Contextual Function Similarity.

GED Graph Edit Distance.

GRUNN Gated Recurrent Unit Neural Network.

GUI Graphical User Interface.

HTTP Hypertext Transfer Protocol.

IDA	Interactive Disassembler.
IM	Instant Messaging.
IO	Input/Output.
IOC	Indicators of Compromise.
IoT	Internet of Things.
IRC	Internet Relay Chat.
IRP	Input/Output Request Packet.
kNN	k-Nearest Neighbours.
LTPL	Linear Temporal Predicate Logic.
MAEC	Malware Attribute Enumeration and Characterization.
ML	Machine Learning.
NP	Non-Polynomial.
OS	Operating System.
QEMU	Quick Emulator.
RAT	Remote Access Trojan.
SEP	Symbolic Execution Point.
SMT	Satisfiability Modulo Theories.
SSL	Secure Sockets Layer.
STP	Simple Theorem Prover.
SVM	Support Vector Machine.
VFS	Virtual File System.
VM	Virtual Machine.

Chapter 1

Introduction

Malware attacks are a significant and frequently reported problem, adversely affecting the productivity of organisations and governments worldwide. Malware development has grown from a fringe hacker activity to a specialised cybercrime activity. Recently, ransomware attacks that encrypt user data and demand payment for decryption have become a substantial problem [1]. The consequences of malware attacks include financial loss, data loss, reputation damage, infrastructure damage, theft of intellectual property, compromise of commercial negotiations, and risks to national security [2].

Reverse engineering is the process of analysis of an artifact in order to obtain knowledge about its design [3]. Malware analysts perform reverse engineering of a malware program to understand a malware samples' capabilities, command interface, relationship with other malware programs, and operating system interactions. The tools used by malware analysts include techniques for the disassembly, detection, clustering, static and dynamic analysis, and similarity of malware samples. This thesis has chosen to focus on studying banking malware and their behaviours, an emulated environment for dynamic malware analysis, techniques to identify similar malware functions, and a technique for the detection of ransomware.

Survey of Banking Malware: Malware authors reuse code, incorporate leaked source code, and build on existing malware products rather than reinventing the wheel. As a result, distinct and persistent malware families have emerged. The literature describing these malware families is fragmented and industry-based. Reliance on commercial literature may not give the full picture as the data may have been lost due to the abandoning of aging webpages, quality is variable, and reports may not be complete as they are driven by commercial interests. So, there is a need for a long term, coherent description of major malware families, their variants, relationships, and information about source code leakages [4]. In this thesis, this need is addressed by studying common banking malware families.

Emulated Execution Environment: Large datasets of historical malware samples are available for countermeasures research. However, due to the age of these samples and anti-virus activities, their original malware infrastructure is no longer available. Dynamic analysis is commonly used for feature extraction; however, due to the absence of the original malware infrastructure, historical malware samples no longer exhibit their original capabilities. Partial execution of historical malware samples in a sandbox results in features that differ from those extracted in the wild, thus invalidating the results of any machine learning based on these features. In this thesis, this need is addressed by studying semi-automatic techniques for the construction of Command and Control (C2) server emulators, and three examples of manual C2 server emulator construction.

Identification of Similar Functions: Internet security organisations perform malware analysis to identify malware capabilities and to develop mitigation techniques. Function similarity is a central technique for malware analysis and is used to determine previously analysed malware samples, to identify updated functions in new malware variants, to exclude unchanged functions in detailed malware analysis, and to identify code reuse across previously unrelated malware families. Function similarity techniques can be used to identify code reuse in malware families and to identify malware products produced by specific malware authors.

Machine learning algorithms can be trained using features from objects and can then classify future instances of these objects in a general manner. Thus an emerging use of machine learning is to detect malware. A less obvious use of machine learning is to identify similar function pairs in malware samples. In this thesis, this need is addressed by the development of function similarity techniques to identify function pairs in different versions of a program.

Ransomware Detection: Ransomware is a widespread class of malware that encrypts files in a victim's computer and extorts victims into the payment of a fee to regain access to their data. Previous research has proposed novel methods for ransomware detection using machine learning techniques. However, this research has not examined the precision of ransomware detection. While existing techniques show overall high accuracy in detecting novel ransomware samples. Previous research does not investigate the discrimination of novel ransomware from common benign programs that share some of the cryptographic characteristics of ransomware. This is a critical, practical limitation of current research; machine learning based techniques would be limited in their practical benefit if they generated too many false positives (at best) or deleted/quarantined critical data (at worst). In this research, this need is addressed by the development of a machine learning based system for ransomware detection and testing of the ability of this technique to discriminate between ransomware and benign cryptographic programs.

Summary: This thesis studies the malware families that attack banking systems and their customers. Techniques are developed for the execution of historical malware families in an emulated C2 environment, techniques for the identification of similar function pairs are developed, and a technique for ransomware identification is developed.

1.1 Motivation

This thesis focuses on the development of components for a malware analysis toolchain. The research highlights the need for techniques based on an analysis of the behaviours of the targeted malware families, and an overarching need to build the research methodology on a solid foundation of malware analysis domain knowledge. Examples of potential problems in research methodology are provided in the chapters dealing with Command and Control (C2) server emulation, and ransomware classification. The identification of similar functions in malware samples assists analysis by allowing the exclusion of functions that have been previously analysed. This allows the identification of new variants, and supports authorship attribution. Machine learning provides a general-purpose technique for the classification of similar objects. A function similarity technique using an SVM model is provided. The unexpected finding is that the SVM model can abstract function similarity features from one pair of program variants to find similar functions in an unrelated pair of program variants. If validated by a larger study, this new property leads to the possibility of creating generic similar function classifiers.

Survey of Banking Malware: Malware authors reuse code, incorporate leaked source code, and build on existing products rather than re-inventing the wheel. This leads to a situation where distinct, persistent malware families have been developed. For example, Gozi malware was first detected in 2007, and the Gozi ISFB variant continues to be used in attacks on banking systems [4]. Prior research has focused on malware detection, clustering, and similarity, but has not focused on specific malware families and the techniques used to manipulate the operating system. This may be due to the view that malware instances are ephemeral and will soon be replaced by new and different malware instances. However, knowledge of persistent and well-established malware families is relevant to academic and industry-based researchers. Different families of information-stealing malware possess similar high-level behaviours and given the finite number of API calls provided by an operating system, this leads to a limited number of options to implement these behaviours. Analysis of the API calls present in a program allows the identification of malware behaviours [5].

Emulated Execution Environment: Large datasets of historical malware samples are available for countermeasures research. However, due to the age of these samples, their Command and Control (C2) servers are no longer available [6]. Dynamic analysis is commonly used for feature extraction; however, due to the absence of their C2 servers, after initialization, malware samples may exit or loop attempting to establish C2 server connections, and as a result, no longer exhibit their original capabilities. Partial execution of historical malware samples in a sandbox results in features that differ from those extracted in the wild, thus invalidating the results of any machine learning using these features. Our approach to extracting accurate features is to build an emulated C2 server to provide an environment that allows control of the full capabilities of the malware in an isolated environment.

Identification of Similar Functions: Internet security organisations identify Indicators of Compromise (IOC) of malware infections and perform detailed malware analysis to identify the malware's command protocol and capabilities [7]. Function similarity techniques are widely used by organisations performing malware analysis, and are used for malware triage [8], program patch analysis [9], identification of library functions containing known bugs [10, 11], identification of similar function pairs in detailed malware analysis [12], malware authorship analysis [13], and for plagiarism analysis [14].

These organisations process large daily feeds of malware samples. Due to packing, these malware samples initially appear to be unique. However, after unpacking, malware sample volumes may be reduced by orders of magnitude by the use of malware similarity techniques, which reduce the malware feed to a smaller set of unique malware payloads [15, 16]. Function level similarity can then be performed on the unique malware payloads to identify known malware families and new variants.

Following the identification of new malware variants, detailed analysis is required to determine their capabilities. This is a time consuming and often manual process that is performed to identify the malware command protocol and capabilities. Function similarity provides the first stage of this analysis and allows the elimination of unmodified functions from the reverse engineering workload [17].

Software vendors periodically provide program patches for security purposes. High-level details of patched vulnerabilities may be provided, but this may not be sufficient for some organisations. Program patch analysis uses function similarity methods to identify updated functions. This is the first step in a reverse engineering process to identify the patched vulnerabilities [18].

Internet of Things (IoT) devices, including routers and consumer electronics, utilize open-source software and a wide range of Central Processing Unit (CPU) architectures. When software vulnerabilities are identified in open source libraries, a wide range of IoT devices become vulnerable. Cross architecture bug search is performed on IoT firmware using function similarity methods that are optimised for searching the same version of source code that has been compiled with different compilers and CPU architectures [10, 19].

Law enforcement anti-malware efforts prioritise identification of malware authors. Function similarity techniques are used to identify code reuse in malware families to identify malware products produced by specific malware authors [13]. Function similarity methods are also used for the analysis of compiled programs in cases where software plagiarism or breach of licensing conditions is suspected [14].

Ransomware Detection: Ransomware is a widespread class of malware that encrypts files in a victim's computer and extorts victims into the payment of a fee to regain access to their data. Previous research has proposed novel methods for ransomware detection using machine learning techniques [20, 21, 22, 23, 24]. However, this research has not examined the precision of ransomware detection. While existing techniques show a high overall accuracy in detecting novel ransomware samples, previous API profile-based research does not investigate the discrimination of novel ransomware from benign cryptographic programs. This is a critical, practical limitation of current research; machine learning based techniques would be limited in their practical benefit if they generated too many false positives (at best) or deleted/quarantined critical data (at worst).

1.2 Research Objectives and Methodologies

1.2.1 Review of Banking Malware Behaviours

Research Objective 1: Study common banking malware behaviours and use static analysis to identify the malware techniques that implement these behaviours.

Banking malware is a class of information-stealing malicious software that targets the financial industry. Banking malware families have become persistent with new versions released by the original authors or by others using leaked source code. The literature describing these malware families is fragmented and industry-based. Problems which arise from reliance on commercial literature is that data may be lost due to the abandoning of aging webpages, quality is variable, and reports may not be complete as they are driven by commercial interests. There is a need for a long term, coherent description of major banking malware families, their variants, relationships, and source code leakages.

Malware behaviour can largely be defined in terms of the API calls made by the execution of a malware sample [25]. Malware behaviours are aggregated into high-level malware capabilities. However, there is no clear definition of all high-level malware capabilities. One of the goals of threat intelligence is to share information about malware attacks. This includes a high-level definition of malware behaviours. Malware behaviours are defined as high-level capabilities by existing standards such as Malware Attribute Enumeration and Characterization (MAEC) [26], but these are broad in scope, and some individual capabilities are not well-defined. There is a need for a well-defined specification of high-level malware capabilities.

The behaviours present in a malware sample may be determined, either by dynamic analysis through execution in a sandbox, or by static analysis of an unpacked malware sample. Static analysis refers to analysis that is performed without executing the program, while dynamic analysis refers to analysis that is performed by executing the program. Static analysis can analyze all program paths; however, obfuscated or encrypted code hinders static analysis [27]. Dynamic program analysis uses program execution to understand it's behaviours and actions [28]. Dynamic analysis is not impacted by the code obfuscations that may hinder static analysis. However, not all program paths are executed, and the program may detect the analysis environment, and alter its behaviour, or terminate [29]. A static analysis technique was selected for the identification of malware behaviours of unpacked code. Thereby providing an analysis of all program code, and preventing any detection of the analysis environment. Malware manipulates the Operating System (OS) using techniques based around the abuse of API calls. The conceptual distance between the low-level details of API calls and a high-level understanding of malware behaviour is known as the semantic gap [25]. One of the challenges of malware analysis is to collect the low-level details obtained through analysis and to make use of these details to synthesise a high-level view of the capabilities of specific malware samples.

1.2.2 Command and Control Server Emulation

Research Objective 2: Development of emulated C2 servers using manual methods, and a review of semi-automated techniques for emulated C2 server construction.

Many types of malicious software are controlled by the attacker's Command and Control (C2) servers [30]. Anti-virus organisations seek to defeat malware attacks by requesting the removal of the C2 server Domain Name System (DNS) records. As a result, the lifespan of most malware samples is relatively short. Large datasets of historical malware samples are available for countermeasures research. However, due to the age of these malware samples, their C2 servers are no longer available. To cope with high volumes of malware production, malware analysis is increasingly performed using machine learning techniques. Dynamic analysis is commonly used for feature extraction. However, due to the absence of their C2 servers, after initialization, malware samples may exit or loop attempting to establish C2 server connections, and as a result, no longer exhibit their original capabilities. Therefore, partial execution of historical malware samples in a sandbox results in features that differ from those that would be extracted in-the-wild, thus invalidating the results of any machine learning research based on these features. One approach to extracting accurate features is to build an emulated C2 server to provide an environment that allows control of the full capabilities of the malware in an isolated environment. Building an emulated C2 server is a complex process, currently this can only be performed by manual or semi-automated techniques [6].

1.2.3 Identify Evolved Function Pairs

Research Objective 3: Development of a technique to identify similar functions in pairs of malware variants that differ by several versions.

Function similarity methods are used for malware triage, program patch analysis, the identification of library functions containing known bugs, malware authorship analysis, identification of similar function pairs in detailed malware analysis, and for plagiarism analysis. Malware authors are known to reuse existing code. This development process results in software evolution and a sequence of versions of a malware family containing functions that show a divergence from the initial version. Given that many malware families have become persistent over a relatively long period of time, the functions in some of these malware variants may no longer show obvious relationships. The challenge in identifying evolved malware function pairs lies in identifying features that are relatively invariant across evolved functions. One of the functions being compared may have been modified in an arbitrary manner, and may no longer provide an obvious match to the original function. Function similarity ground truth may be established by reverse engineering malware variants and identifying the corresponding function pairs.

1.2.4 Identify Function Pairs using Machine Learning.

Research Objective 4: Development of a machine learning technique to identify similar functions in a pair of malware samples.

The identification of similar functions in malware samples assists analysis by allowing the exclusion of functions that have been previously analysed. This allows the identification of new variants, supports authorship attribution, and the analysis of malware phylogeny. Machine learning provides a general-purpose technique for the classification of similar objects. The challenge in performing machine learning classification on evolved functions in malware variants is that there may not be obvious similarities between the evolved functions that are being compared. This research objective requires the development of a machine learning system that can identify evolved functions pairs present in two variants of the same malware family. Features for identifying evolved function pairs are present inside the functions that are being compared, but additional contextual features are present in functions that are related to the functions that are being compared. In this study, demonstrate that contextual features stay relatively invariant across different malware versions. Due to the high false positive rates that have been observed in previous machine learning function similarity studies, a two-stage system for identifying malware function pairs is acceptable.

1.2.5 Generalisation of Function Context

Research Objective 5: Development of a function similarity technique using an improved function context to identify similar functions in a pair of unrelated malware families.

Contextual features are function similarity features extracted from specific functions closely related to the functions being compared. The extraction of contextual features can substantially improve the strength of function similarity features. This study should identify methods to further improve the strength of the contextual function similarity features. A measure of the machine learning system's performance can be obtained by training on features extracted from one dataset and performing classification using features extracted from a second independent dataset. Good classification results from an independent dataset indicate that machine learning can abstract the training features. The objective of the research in this chapter is to use the Zeus Support Vector Machine (SVM) model and apply it to features extracted from a pair of ISFB malware variants. Successful prediction of function similarity in the unrelated pair of malware variants will indicate that the machine learning system is successfully abstracting function similarity features.

1.2.6 Discriminate Ransomware from Benign Cryptographic Software

Research Objective 6: Development of a ransomware detection system that can discriminate between ransomware and benign cryptographic software.

Ransomware is a widespread class of malware that encrypts files in a victim's computer and extorts payment to regain access to the encrypted user files. Previous research has proposed methods for ransomware detection using machine learning techniques. While these techniques show an overall high accuracy in detecting novel ransomware samples, this research has not investigated the discrimination of ransomware from benign cryptographic programs.

The research objective is to build a ransomware detection system and test its ability to discriminate between ransomware and benign cryptographic software. The features used in this research should be taken from the dynamic analysis logs of a Cuckoo sandbox. Feature engineering will be used to select the highest performing API call features.

1.3 Contribution

The key contributions of this thesis are as follows:

- Assembling a coherent description of major banking malware families, their variants, relationships, behaviours, and source code leakages.

- Identification that when created, banking malware families become persistent threats, and that knowledge of these malware families is relevant to academic and industry based malware researchers
- Identification of the limitations of extracting features from historical malware samples without the use of an emulated C2 server.
- Development of an emulated C2 environment for dynamic malware analysis.
- Develoment of a machine learning system to identify similar malware functions.
- Use of feature engineering to develop contextual features, this is a method of strengthening function similarity features.
- A study into whether a machine learning based ransomware detection system can confuse ransomware and benign cryptographic programs.

This thesis provides the following additional contributions:

- Identification of a set of key behaviours to allow a high-level comparison of the techniques used to implement banking malware families.
- Identification of the malware techniques that are used to implement the identified banking malware behaviours.
- Demonstration that the Pharos Framework can provide identification of banking malware behaviours using static analysis.
- Creation of three C2 server emulators for the Zeus, Cryptowall and CryptoLocker ransomware families for dynamic analysis.
- Development of Evolved Similarity Techniques (EST), for extracting invariant features associated with malware function pairs that differ due to software development.
- Development of the Cross Version Contextual Function Similarity (CVCFS) technique for finding similar functions in pairs of malware programs.

- Development of a curated dataset of matched functions in three versions of Zeus malware for use in future research.
- Development of a set of three labelled IDA databases of Zeus malware versions 2.0.8.7, 2.0.8.9, and 2.1.0.1.
- Development of the Cross Version Function Similarity Using Contextual Features (CVCFS) technique, a 2-step machine learning based method for the identification of similar pairs of functions in two variants of a program.
- The performance of CVCFS numeric features has been increased through the use of improved feature encoding.
- Development of the Function Similarity Using Family Context (FCSFS) technique, a one-step method for function similarity that performs well without pre-filtering and provides a low false positive rate.
- Development of a curated dataset of matched functions in two versions of ISFB malware for use in future research.
- Development of a set of two labelled IDA databases of ISFB malware versions 2.04.439, 2.16.861.
- Development of a technique to improve the discrimination of ransomware from benign-cryptographic programs.

1.4 Thesis Layout

This thesis presents research contributions in the form of published papers and consists of eight chapters.

- Chapter 1 introduces the need for automated techniques for malware analysis.
- Chapter 2 provides a literature review of research into automated techniques for malware analysis.

- Chapter 3 provides a study of the behaviours of common banking malware families. This work was published by the Computers and Security Journal in 2018.
- Chapter 4 provides a study into problems associated with the dynamic analysis of historic malware samples and proposes the use of C2 emulators to overcome these problems. Work in this chapter has been accepted as a book chapter for Malware Analysis using Artificial Intelligence and Deep Learning, published by Springer in 2020.
- Chapter 5 provides a technique for identifying function pairs in malware variants, and contributions were published by the IEEE Trustcom conference in 2019.
- Chapter 6 improves the function similarity technique from chapter 5 using machine learning and has been submitted to the Computers and Security Journal for review.
- Chapter 7 makes extensive use of contextual feature engineering to provide a substantial improvement to the performance of the function similarity technique. This work was published in the Electronics Journal in 2020.
- Chapter 8 examines the problem of discriminating ransomware from benign programs containing similar cryptographic operations and has been submitted to the Workshop on Artificial Intelligence and Cybersecurity (AICS) ICONIP 2020.
- Chapter 9 provides the conclusion and future work.

Detailed bibliography and related appendices are provided at the end of the thesis.

1.5 Conclusion

This chapter provides an introduction to this thesis and outlines the problems cause by ongoing malware attacks and the need for a study of banking malware families, their behaviours, and the techniques for identification of these behaviours. This is followed by an examination of the need for emulated command and control server for the analysis of historical malware samples. Function similarity is an important and widely used technique in various stages of the malware analysis process. Function similarity techniques help in fast tracking the malware analysis process. This thesis presents techniques that can operate on the same malware family and across different malware families. This is followed by an overview of the need for a technique to detect ransomware that frequently targets banking customers by encrypting user files and demanding payment to decrypt these files. Existing API profile-based research has not examined the discrimination of ransomware from benign cryptographic software. Chapter 2 provides a review of the research literature related to the topics covered in this introduction.

Chapter 2

Literature Review

This chapter provides a review of the literature related to the main themes of this thesis that were identified in chapter 1. These themes are a survey of banking malware and its behaviours, an emulated environment for dynamic malware analysis, techniques to identify similar malware functions, and a technique for the detection of ransomware. A taxonomy of malware function similarity techniques is shown in Figure 2.1

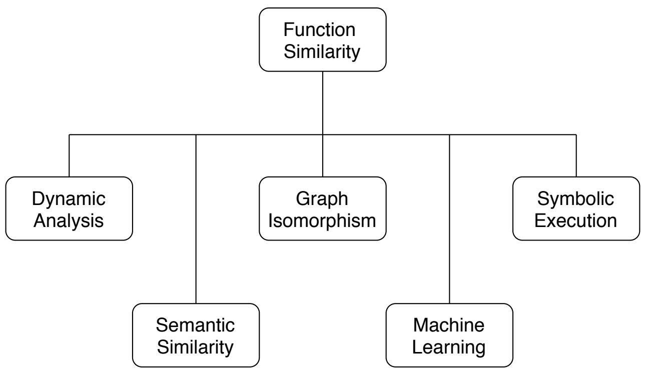


Figure 2.1: Malware Function Similarity Taxonomy

2.1 Malware Behaviour

An OS provides a finite number of API calls to allow application programs to access OS services. Consequently, the set of all possible operations that can be performed by a finite program using OS services is finite. Malware operates by manipulating OS behaviour; therefore, malware operations can be represented as a finite set of API calls [25]. The monitoring of malware API calls using dynamic analysis results in a detailed log. This log of malware API calls needs to be converted into a higher-level representation that would be more easily understood by an analyst. The conceptual distance from a low-level log of API calls to a higher level representation is known as a semantic gap. Martignoni et al [25] bridge this semantic gap by mapping the high-level goals of the malware into finite sets of API calls. The set of high-level goals is referred to as the malware's behaviour [25].

Dynamic analysis of malware behaviour has been used to extract specifications of malware behaviour [31]. The specifications are derived from API call profiles obtained from dynamic analysis. These malware specifications are used for the detection of malware. Malware capabilities labelling, using Linear Temporal Predicate Logic (LTPL) operating on a dynamic trace of file system access, has been used to identify malware capabilities as part of a malware detection system [32].

The malicious behaviours of malware are modelled by using tuples of Subject, Object, Action, and Function, where subject represents active system entities, object is a system resource which is used to store information, action is the malicious behaviour which is initiated by a trigger, and function is the outcome of the action performed by a subject on an object [33].

An extensible malware taxonomy is built from a set of malware behaviours grouped into the following classes: Evasion, Disruption, Modification, and Stealing. The Evasion class contains behaviours pertaining to the removal of evidence, removal of registries, anti-virus engine termination, and firewall termination. The Disruption class includes behaviours for the scanning of known vulnerable services, email sending, Internet Relay Chat (IRC)/Instant Messaging (IM) known port connection, and IRC/IM unencrypted commands. The Modification class contains behaviours pertaining to the creation of a new binary, creation of unusual mutex, modification of name resolution file, modification of browser proxy settings, modification of browser behaviour, persistence, download of known malware, download of unknown files, and driver loading. The Stealing class contains behaviours pertaining to the theft of system/user data, credential or financial data theft, and process hijacking [34]. It is noted that the behaviours identified by Grégio et al. are not complete, for example, gaining control of a command shell or desktop (Backconnect) could be considered to be a member of the Modification class [34].

2.2 Emulated Execution Environments

Research techniques exist for automatic protocol analysis of malware [35]. However, these techniques depend on malware communications with live C2 servers. The usage of the malware capabilities is determined by the malware operators, and live testing may not reveal the full extent of the malware's capability. Other issues related to performing research with live malware include difficulties in obtaining a consistent supply of live malware, unknown configuration, unknown triggering conditions, detection of the analysis IP address (mitigated by the use of an anonymizing proxy), or the malware operators gaining access to the analysis VM via malware provided interfaces.

Researchers have recognised the need to prevent malware experiments from causing harm on the internet. Research systems have been built to provide containment of malware research [36]. However, these systems do not address the C2 server problems faced when performing experiments with historical datasets. Internet simulator programs [37] may be used as part of a malware analysis environment and can provide generic responses to requests for common internet services. A malware process may request a connection to a common website to perform a connectivity check, and an internet simulator may be able to satisfy this request. However, if a connection to a C2 server or other attacker-controlled infrastructure is requested, an internet simulator will not be able to respond with the protocol required by the malware.

The Botnet Evaluation Environment provides an isolated environment for botnet research with emulated C2 servers for execution of the Agobot, SDBot, GTBot, Phatbot, and Spybot malware [38]. An isolated Waledac botnet was created by reverse engineering the Waledac malware and identifying the Waledac botnet protocol. An emulated C2 server was built to support this protocol, and a 3000 node Waledac botnet was built. This isolated botnet was used to research security vulnerabilities that could be used to take down the Waledac botnet [39].

While the ability to automatically generate C2 server emulators for arbitrary malware families would be useful, this is not currently feasible, and the recent work in literature is a semi-manual construction process.

The Imaginary C2 program [40] converts captured network traffic into request definitions that allow C2 HTTP response to be replayed. However, this traffic replay approach is not suitable for situations where initial network traffic samples are not available.

One automation approach for the creation of C2 server emulators is provided in [41]. This research refers to C2 server emulators as Custom Impersonators. Malware samples are executed on a Quick Emulator (QEMU) Virtual Machine (VM), and instruction traces are collected using DECAF [42], and the instruction traces are translated into VINE intermediate language [27]. Symbolic execution [43] is performed on the instruction traces, and symbolic variables are assigned to network input. A Simple Theorem Prover (STP) constraint solver [44] is used to determine the values that determine the outcome of the control flow tests. These values can be used to identify malware control dependencies controlled by values in the network input [41]. The malware control flow graph and control dependencies are provided to assist analysts with the manual construction of C2 server emulators.

Research using ANGR [45], an open source symbolic execution framework, creates a technique that employs static analysis to determine the C2 command protocol and associated commands implemented in a common Remote Access Trojan (RAT). The top-level command processing function of the RAT is analysed, and for each explored path, a list of the malware API calls and their arguments, function call relationships, and the network data required to trigger the path's execution are provided [46]. Windows API models and support for the `stdcall` calling convention were added to ANGR in order to support the analysis of Windows malware. Heuristics were created to limit the number of paths explored by the symbolic execution to prevent potential path explosion problems. Symbolic execution commences at the manually selected Symbolic Execution Point (SEP), an execution context is needed to provide precondition values that are generated in malware initialization. In this research, the execution context was generated using two different techniques: by performing concrete execution, setting a breakpoint at the SEP, taking a memory dump and extracting the necessary parts of the execution context, or by moving the SEP backward, allowing initialization of execution context values. Symbolic execution was used to explore the command processing loop.

The report produced by this technique showed the API calls, and the functions called in processing each command as well as the network data required to trigger the processing of each command. This research targets analysis for a single RAT, requires manual SEP identification, does not support the analysis of encrypted protocols, and does not provide support for mining the analysis report from the tool output [46].

The S2E symbolic execution engine is the basis of research that constructs C2 servers for RAT [6]. The S2E engine performs symbolic execution of instructions and forks execution when branches are taken. A Satisfiability Modulo Theories (SMT) solver is used to evaluate expressions and obtain concrete values. To prevent performance problems and scalability issues due to path explosion, an analyst is required to provide the location of the command processing loop and details of how to reach this address. The process used in this research can be summarised as Trace Generation, Trace Analysis, Speculation, Validation, and C2 Server Generation. Trace generation uses symbolic execution to explore execution paths and to maximize code coverage. A number of the recorded traces will cover the RAT command processing code. The branches taken and API execution details are recorded. Trace analysis builds Augmented Prefix Tree Acceptors (APTA) that captures API execution and branches taken along the explored paths. APTA are Deterministic Finite Automata (DFA) that have been used in the protocol reverse engineering [47]. The goal of speculation is to generate a small number of paths that covers all of the commands. Speculative edges are added to the APTA in an attempt to combine symbolically executed command fragments into paths containing multiple commands. The symbolic execution engine is then used to validate speculatively generated paths when speculative edges are validated; the branches and API calls are recorded. C2 server generation is performed for each validated path that contains multiple commands. This research generates a C2 server from the code of a small RAT created for research purposes [6]. Due to the requirement for manual analysis to provide the location of the command processing loop as a starting point, this research is classified as semi-automated.

2.3 Function Similarity

The purpose of function similarity is to measure the degree of similarity in the functions in two compiled programs. The techniques used in the research literature make use of graph isomorphism, symbolic execution, semantic similarity, dynamic analysis, and machine learning. Cross architecture bug search is a specialization of function similarity that aims to identify vulnerable library functions that have been created for a range of IoT devices that utilize different compilers and CPU architectures.

2.3.1 Graph Isomorphism

One of the first applications of function similarity was in the analysis of vendor security patches for compiled programs and libraries [9, 18]. This approach constructs a Control Flow Graph (CFG) for each program, identifies basic blocks and functions. Graph isomorphism and edge count are used to identify basic block and function similarity. This structural approach to function similarity does not depend on the contents of each function, which may have been modified by code updates, or changed compiler settings. Instead, it relies on the CFG being relatively unchanged by program updates. This research was used to develop the commercial BinDiff program [48].

BinDiff provides a graph-based analysis of the differences between two versions of the same program. It was designed to determine the program changes introduced by security patches [49]. BinDiff uses program structure and syntactic features such as string references when calculating similarity [12]. BinDiff identifies the functions in the two programs and calculates the Control Flow Graph (CFG) [50] of the functions and uses graph isomorphism to identify function pairs. Metrics such as edge counts, derived from the CFG are used to estimate function similarity. BinDiff initially calculates unique function matches. Additional matches are then found by searching for more unique matches from unmatched neighbouring functions [51]. The BinDiff user interface displays a ranked list of matching function pairs, a similarity ratio, and a confidence statistic. BinDiff displays the CFG of selected functions and identifies matching, differing, and removed basic blocks. BinDiff operates well when the programs being compared exhibit similar CFGs, however recompiling a program with a high optimisation setting changes the CFG sufficiently to reduce Bindiff's accuracy to 25 percent [52].

BinSlayer determines pairwise function similarity between malware samples. It uses the BinDiff algorithm and the Hungarian Algorithm for bi-partite graph matching. Graph Edit Distance (GED) defines the minimum number of edit steps required to transform graph A to graph B. GED is a Non-Polynomial (NP) hard problem, but may be approximated by transforming the problem into a bipartite graph matching problem which may be solved using the Hungarian Algorithm. The matching provided by BinDiff is brittle, and can fail to match functions that are similar but not identical. The aim of BinDiff is to match functions that are identical at an abstract level. The aim of BinSlayer is to match functions which are similar. The concept of matching similar functions is encapsulated by the shortest GED metric. Supplementing the BinDiff Algorithm by use of the Hungarian Algorithm allows the matching of functions that are close to identical as well as functions which are similar.

2.3.2 Symbolic Execution

BinHunt identifies matching functions in two programs and extends earlier work by constructing the CFG for each program, it uses symbolic execution to identify matching basic blocks, theorem proving is used to test whether the basic blocks are semantically equivalent. Graph isomorphism is used to identify matching functions [53]. BinHunt accounts for register renaming by testing all permutations of registers and variables. As a result, BinHunt requires significant computing resources [51] [49].

Expose performs identifies similar functions in compiled programs but does not specifically target malware function similarity. The first step in Expose processing is a pre-processing step that calculates a matching score based on input parameter count, out-degree, function size, and cyclomatic complexity. The pre-filtering step excludes loader functions and unlikely matches. Expose uses n-gram analysis at the function level to identify functions that are not suitable for symbolic execution. Symbolic execution is used with a theorem prover to establish semantic matching. For functions, where a semantic match cannot be calculated, Expose calculates syntactic function similarity and tests call relationships. Syntactic function similarity uses n-grams of disassembled code and cosine distance to identify the remaining function pairs [54].

2.3.3 Semantic Similarity

A problem with graph isomorphism based approaches is that this technique does not scale well, and some use cases seek to identify similar functions in large datasets. The use of feature hashes avoids the performance issues associated with graph isomorphism. One approach uses the minhash of the Input/Output (IO) behaviour of basic blocks in order to cluster similar functions in a large malware dataset [55]. BinJuice computes abstract semantics for each basic block of all functions in a compiled program, by a process of disassembly, CFG generation, symbolic execution, algebraic simplification, and the computation of function semantics [49]. Four different representations of the unpacked code are created, these are disassembled code, generalised code, semantics, and generalised semantics (juice). VirusBattle is built on BinJuice and is used to identify relationships between malware samples in a large database [56]. VirusBattle identifies similar functions by comparing the hashes of the function's generalised semantics. This provides a significant performance gain compared with non-approximate methods, e.g., theorem solvers [57].

BinJuice extracts abstract semantics, called juice, for each basic block of all functions in a compiled program. The aim of generating abstract semantics is to represent any two equivalent code sequences by the same semantics. BinJuice computes abstract semantics for each basic block of all functions in a compiled program, by a process of disassembly, CFG generation, symbolic execution, algebraic simplification, and the computation of function semantics [49]. Four different representations of the unpacked code are created, these disassembled code, generalised code, semantics, and generalised semantics (juice). Disassembly and CFG extraction are performed by existing tools. Basic block semantics are generated using symbolic interpretation. Symbolic execution does not involve execution on a physical processor; instead, the effects of the program instructions can be represented as a set of simultaneous equations. An algebraic simplifier provides simplification of the symbolic expressions resulting from the symbolic execution. The simplified symbolic expressions are mapped into a canonical form to ensure that sections of equivalent code having equivalent symbolic expressions are sorted in the same order.

The compilation process that generates equivalent functions will not be expected to use the same register names or the same memory addresses. To facilitate the identification of equivalent functions generalisation of code and semantics is performed by replacing register names and memory addresses with symbolic values. The instruction `mov(eax,dptr('0x3004'))` would be represented in gen_code as `mov(A,B)`, in semantics as `A=pre(memdw('0x3004'))`, and in gen_semantics as `E=C`. This generalisation of semantics yields an abstract semantics that is resistant to code variation due to compiler settings [49]. Logic variables are generated in a consistent manner, so that different register and memory selections will be reduced to consistent logic variables. [54, 49, 57].

VirusBattle is built on BinJuice and is designed to identify relationships between malware samples in a large dataset [56]. VirusBattle identifies similar functions by comparing the hashes of the function's generalised semantics. This provides a significant performance gain compared with non-approximate methods, e.g., theorem solvers [57]. Basic block similarity can be established by string comparison of the hash of the block's juice and does not require the use of a theorem prover, thereby obtaining a significant performance gain [57]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

2.3.4 Dynamic Analysis

Blanket Execution is a dynamic analysis technique for identifying similar functions, by executing function pairs in the two programs being compared. A set of pseudo-random inputs are provided for each function pair. Features are taken from the following values: reads or write to the heap, reads or writes to the stack, calls to library function, system calls, the function return value. Blanket execution has an accuracy of approximately 77%. An advantage of this technique is that its performance is independent of compiler toolchain, and optimization options [52].

2.3.5 Cross-Architecture Bug Search

The proliferation of IoT devices using open source code and a variety of CPU architectures has led to a research interest in identifying defects in the same source code created with different compilers for several different CPU architectures. These techniques are optimised for cross-architecture vulnerability identification within one version of a program.

DiscovRE [10] identifies vulnerable functions across a range of IoT devices. IoT devices often use open-source code and a variety of CPU architectures. Vulnerability discovery in an open-source library may impact a range of IoT devices. The identification of functions containing known vulnerabilities requires the capability to search code generated by different compilers, and a range of CPU architectures. DiscovRE uses k-Nearest Neighbours (kNN) machine learning to pre-filter potentially vulnerable functions. The Pearson product-moment correlation coefficient [58] has been used to select numeric features that are robust across multiple compilers, compiler options, and CPU architectures. The following function counts have been used as features: call instructions, logic instructions, redirection instructions, transfer instructions, local variables, basic blocks, edges, incoming calls, and instructions. The final identification of vulnerable functions is performed using maximum common subgraph isomorphism.

In some scenarios, the DiscovRE pre-screening stage is unreliable [19]. CVSkSA performs cross-architecture searching for vulnerable functions in IoT device firmware using the kNN algorithm, followed by SVM machine learning for pre-filtering [19]. Bipartite graph matching [11] is then used to identify vulnerable functions from the pre-screened matches. Although the accuracy of the SVM model for pre-filtering is good, the run time is slow, but kNN pre-filtering reduces the number of functions to be checked by the SVM model, and this reduces execution time by a factor of four with a small reduction in vulnerability identification performance. CVSkSA uses two levels of features, function level features are used in the pre-filtering while basic block level features are used for graph matching. The function level features are call count, logic instruction count, redirection instruction count, transfer instruction count, local variables size, basic block count, incoming call count, and instruction count.

Gemini [59] and SAFE [60] use neural networks for cross-architecture bug-search, as these techniques exhibit better performance characteristics than graph isomorphism. Gemini performs cross-architecture, binary, software defect search using CFG features. Gemini uses a neural network to extract features from the CFG of the compiled functions [59].

SAFE extracts the instruction sequence from compiled functions and models them as a natural language [60]. The interaction of the instruction sequence is captured using a Gated Recurrent Unit Neural Network (GRUNN). An attention mechanism is used to automatically focus on the instructions with the best performance for the identification of similar functions.

2.4 Dynamic Analysis for Ransomware Detection

Existing research deals with detecting ransomware using machine learning [20, 21, 22, 23, 24, 61]. *EldeRan* [20] demonstrates the importance of feature selection to reduce the overall complexity of the problem and to improve the performance of machine learning. *EldeRan* uses features from the following classes: API calls, registry key operations, file system operations, file operations per file extension, directory operations, dropped files, and strings. The dataset used in this research consisted of 582 samples of ransomware belonging to 11 malware families and 942 benign programs. The benign programs consisted of generic utilities for Windows, drivers, browsers, file utilities, multimedia tools, developer's tools, network utilities, paint utilities, databases, emulator and virtual machine monitors, office tools. While this is a comprehensive dataset, it does not specifically target programs with cryptographic features that could be misclassified as ransomware. Experiments were performed to test the ability of *EldeRan* to detect known ransomware, and also to detect novel ransomware. Testing with known ransomware provided an average accuracy of 0.963 and testing with novel ransomware samples gave a detection rate of 0.933 with 100 features and a detection rate of 0.871 with 400 features [61].

The research in [23] uses Windows API call data from the Cuckoo sandbox to generate a vector model of API calls to train an SVM machine learning model for ransomware detection Two different vector representations are generated; the first is a vector representation that encoded the API call logs using a q-gram frequency, and a standardized vector representation. The research uses 312 samples of benign software; further details of these programs are not provided. 276 ransomware programs targeting the Windows Operating System are used in this research. This dataset includes "WannaCry, Cerber, Petya, CryptoLocker, and so on", but further details are not provided. The accuracy of this research using the proposed vector format was 0.9352, and 0.9748 using the extension to standardized vectors technique. The published results do not include true positive or false positive values. It is noted that the data was not divided into malware families before testing, this allows the possibility that samples of malware families present in the training data were also present in the test data, raising the apparent average detection accuracy.

RansHunt is a hybrid analysis system that used static and dynamic analysis for ransomware detection. RansHunt uses the following feature classes: function length frequency, strings, API calls, registry operations, process operations, and network operations [61]. The dataset used in this research consisted of 360 samples of ransomware from 21 families, 532 different types of malware, and 460 benign software. Details of the types of benign software in the dataset were not provided. A 10 fold cross-validation approach was used in this paper. The selection of a given number of ransomware families for training and using the remaining families for testing would give a more accurate understanding of the performance of this research. Feature selection is performed using Mutual Information criteria. The accuracy and precision values for static analysis were 0.935/0.951, for dynamic analysis was 0.961/0.960 and were 0.971/0.970 using the hybrid approach. [61] analyses network traffic in order to detect ransomware by monitoring DNS resolution requests to specific black-listed domains. They detect dynamically generated C2 servers domain names to identify Secure Sockets Layer (SSL) key exchange messages.

An analysis of the API calls made by malware samples from 14 malware families concluded that it may be feasible to identify ransomware behaviour on the basis of API call profile data alone [22].

CryptoDrop is able to detect ransomware by monitoring changes to user files [62]. This is performed by monitoring changes to the following: file similarity, file type, file entropy, and file deletion operations.

GURLS [24] uses API call frequency features and machine learning based on regularized least Squares for ransomware detection. The highest average binary detection rate of 0.886 was achieved using a radial basis kernel. Multiclass classification was used to identify each ransomware family with an average accuracy of 0.867.

2.5 Conclusion

The identification and analysis of malware is an important and challenging task. This chapter has reviewed a number of promising techniques for automating this process, but several open research questions remain.

The first question relates to the nature of malware. Research has primarily focused on techniques for malware identification, clustering, and similarity, but does not focus on the nature of individual malware. This leaves an implied assumption that existing analysis techniques are best suited for their purpose, but without an analysis of the behaviours and malware techniques of existing and new malware families, this assumption cannot be verified.

The second question relates to the dynamic analysis of historic malware samples. The execution of historic malware samples which no longer have active C2 servers results in the execution of the malware initialization functions. It does not provide the execution of the full malware capabilities. The creation of an emulated C2 server allows the control of the malware sample and the execution of its full capabilities. Research into the creation of C2 server emulators is in its early stages.

The third question relates to the identification of similar function pairs in two programs. Graph isomorphism techniques have been used for this purpose but do not scale well as the number of functions in the programs increases. Symbolic execution and theorem proving techniques perform well to identify closely matched functions but do not perform well for general function similarity. Machine learning techniques have been used in Cross Architecture Bug Search research. Machine learning in this application has been used as a pre-filter due to the high false positive rate. There is a need for a machine learning technique that can identify function similarity with high accuracy across a wide range of programs without requiring retraining.

The fourth question relates to the detection of ransomware. While machine learning techniques for the detection of ransomware have been developed, there has been no assessment of these techniques' ability to discriminate ransomware from benign cryptographic programs. This is an important question for the practical application of machine learning techniques to ransomware protection for computers at large.

Chapter 3

A Survey of Similarities in Banking Malware Behaviours

This chapter explores knowledge specific to banking malware, their history, families, behaviours, source code leakages, and inter-relationships. A review of contemporary banking malware families is performed, and it is found that the literature is fragmented, industry-based, and provided to fulfill commercial goals. This paper draws together a coherent description of major banking malware families, identifies malware behaviours present in these families, and details the implementation of these behaviours. Banking malware families share common high-level behaviours, and an analysis of these behaviours shows that their implementation follows specific patterns of abuse of operating system facilities. This chapter assembles a dataset of malware behaviour details and then uses the Pharos Framework to bridge the semantic gap between low-level details and high-level behaviours.

The work in this chapter has been published as a journal paper:

- P. Black, I. Gondal, and R. Layton, "A Survey of Similarities in Banking Malware Behaviours," *Computers & Security Journal*, vol. 77, Impact factor 3.579, 2018.

Available online at www.sciencedirect.com

ScienceDirect

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

Computers & Security

A survey of similarities in banking malware behaviours

Paul Black[*], *Iqbal Gondal*, *Robert Layton*

Internet Commerce Security Lab, Federation University, Australia

ARTICLE INFO

Article history:
Received 30 August 2016
Revised 25 August 2017
Accepted 21 September 2017
Available online 9 October 2017

Keywords:
Malware analysis
Malware components
Banking malware
Malware similarity
Malware capabilities

ABSTRACT

Banking malware are a class of information stealing malicious software that target the financial industry. Banking malware families have become persistent with new versions being released by the original authors or by others using leaked source code. This paper draws together a fragmented and industry based literature to provide a coherent description of major banking malware families, their variants, relationships and source code leakages. The concept of malware behaviour is well established in the research literature. However, the literature has not settled on an identification of key malware behaviours. Malware behaviours are defined by existing standards, but they are broad in scope and some individual behaviours are not well defined. This paper identifies a set of malware behaviours that are present in the selected banking malware families. The conceptual distance between the low level detail of Application Programming Interface (API) calls and a high level understanding of malware behaviour is known as the semantic gap. This paper assembles a dataset of malware behaviours and then shows experimental use of the Pharos Framework to bridge this semantic gap by providing automatic identification of malware behaviour using static methods.

1. Introduction

This paper uses the term “banking malware” to refer to information stealing malware developed by criminal groups, which are used to attack the financial industry. Banking malware are known to attack Microsoft Windows and other operating systems. An industry report states that in 2016 approximately 70 percent of malware samples targeted the Windows Operating System (OS) (AV-Test, 2017). A significant body of research, analysis, and reporting exist for malware attacking the Windows OS. The malware considered within this paper is specific to the Windows OS, however the method described in this paper is applicable to malware targeting any OS.

Current literature describing malware families is largely produced by commercial organizations. Problems which arise from reliance on a commercial literature is that data may be lost due to the abandoning of aging webpages, quality is variable and reports may not be complete as they are driven by commercial interests. Malware research has focused on techniques for malware detection, classification, clustering and malware similarity (Mohaisen et al., 2015). Prior research has not focused on specific malware families and the evolution of these malware families. This may be due to a view that malware instances are ephemeral and are soon replaced by new and different malware instances. This paper shows that these banking malware families have become persistent, are evolving and that knowledge of these malware families is relevant to academic and industry based malware researchers.

Dynamic analysis has been widely used to obtain an instruction trace or a trace of API calls. A limitation of dynamic analysis is that it is rarely possible to obtain full execution

[*] Corresponding author.
E-mail addresses: paulblack@students.federation.edu.au (P. Black), iqbal.gondal@federation.edu.au (I. Gondal), robertlayton@gmail.com (R. Layton).

https://doi.org/10.1016/j.cose.2017.09.013
0167-4048/© 2017 Elsevier Ltd. All rights reserved.

coverage of a malware sample. Static analysis can be used to extract a graph of API calls from unpacked malware samples. Static analysis has the advantage of providing full coverage of the malware sample but may have difficulties determining API calls made using obfuscation techniques. The API calls details obtained using either dynamic or static analysis methods have previously been used for malware detection and malware classification.

Although many different banking malware families exist, they share the same high-level goals, that is to enable information stealing and financial crime. The aim of this paper is to examine similarity in the behaviours of this selected set of malware which serve similar purposes but have different and somewhat unknown origins. This paper provides an example of the use of the Pharos Framework to identify behaviours which are present in a set of banking malware samples. This automatic classification of the behaviours present in a malware sample presents a representation of malware capability at a higher level of abstraction than has previously been available.

The requirement for the identification and comparison of malware behaviours is different from the comparison of similarity between malware functions. To establish similarity between functions it is necessary to show that the actions performed by two functions are equivalent. To compare the behaviours of two malware samples, it is necessary to show that the aggregate behaviour of a group of functions in each sample is equivalent.

A malware family comprises all the variants of a malware containing common malicious behaviours (Christodorescu et al., 2005). To understand the relationship between banking malware and other types of malware, attempts have been made to build a malware taxonomy (Grégio et al., 2015). However these categorisation schemes have been outgrown by the rapid proliferation of new types of malware.

The structure of this paper is as follows, the literature review examines the concept of malware behaviour, then a review of the selected malware families is given. This is followed by an overview of the different ways that the selected malware behaviours may be implemented. The implementation details of the behaviours of each malware family is given. Finally, an example of the static identification of malware behaviours is provided by the use of the Pharos Framework.

This paper provides a survey of the behaviours of the following selection of recent Microsoft Windows banking malware families: Zeus V2, Citadel, Carberp, Vawtrak, Dridex, Dyre and Rovnix malware. These malware families were selected based on prevalence. Table 1 shows a timeline of banking malware families, the inheritance relationships between variants and the dates of source code leaks.

Table 1 – Banking malware relationships.

Malware	Detected	AKA	Variant	Source Leak
Atmos	2015		Zeus	
Banjori	2013	Multibanker BankPatch		
Bedep	2014	Rozena		
Bolek	2016	Kbot		
Buhtrap	2014			2015
Carberp	2010			2013
Carbanak	2015	Anunak	Carberp	
Chthonic	2014		Zeus	
Citadel	2012		Zeus	
Corebot	2015			
Cridex	2011	Feodo Bugat		
Dridex	2014	Bugat Geodo	Cridex	
Dyre	2014	Dyreza Dyzap Dyranges		
Fobber	2015		Tinba	
Gameover Zeus	2011	Zeus P2P	Zeus	
Gozi	2007	Snifula Ursnif		2010
GozNym	2016		Gozi Nymaim	
ICE IX	2011		Zeus	
ISFB	2015	Ursnif Papras	Gozi	2015
Kins	2011	VM Zeus Zberp Zeus		
Kronos	2014			
Panda	2016		Zeus	
Qadars	2013			
Qakbot	2009	Qbot		
Ramnit	2010			
Ranbyus	2012			
Rovnix	2011		Carberp	
Shiz	2006	Shifu		
Tinba	2012	Tinybanker Zusy		2014
Trickbot	2016	TrickLoader		
Urlzone	2009	Bebloh Shiotob		
Vawtrak	2013	Neverquest	Gozi	
Zeus	2006	Zbot		2011

1.1. *Evolution and persistence*

Once created, banking malware families have become persistent and have continued to evolve, despite opposition from law enforcement. This persistence is due to the malware author's reuse of source code, the incorporation of leaked malware source code and a desire to build on existing malware products rather than re-inventing the wheel. An example of the persistence of banking malware families is provided by the Zeus malware. Zeus version 1 was detected in 2006. Zeus version 2 was detected in 2010. Gameover Zeus was detected in 2011. The Zeus V2 source code was leaked in April 2011 (Baumhof and Shipp, 2011; Riccardi et al., 2011). Following the Zeus source code leak Citadel, ICE-IX, Kins, Panda and Chthonic Zeus variants were developed in a period spanning from 2011 untill 2016. Leaked source code provides an opportunity for a malware analyst to increase the insight of their analysis, however leaked source code also provides an opportunity for malware authors to improve their products.

1.2. *Contribution*

The contributions of this paper are as follows:

- Assembling a coherent description of major banking malware families, their variants, relationships and source code leakages.
- Identification that when created banking malware families are persistent with new versions being released by the original authors, or by others using leaked malware source code.
- Identification of a set of key behaviours that allow a high-level comparison of the techniques used to implement banking malware families.
- Identification of the techniques that are used to implement the identified behaviours.
- Demonstration of the Pharos Framework which can provide behaviour identification by static analysis.

2. Related work

An OS provides a finite number of API calls to allow application programs to access OS services. Consequently, the set of all possible operations that can be performed by a finite program using OS services is finite. Malware operates by manipulating OS behaviour; therefore, malware operations can be represented as a finite set of API calls (Martignoni et al., 2008). The monitoring of malware API calls using dynamic analysis results in a detailed log. This log of malware API calls needs to be converted into a higher level representation that would be more easily understood by an analyst. The conceptual distance from a low level log of API calls to a higher level representation is known as a semantic gap. Martignoni et al. (2008) bridge this semantic gap by mapping the high-level goals of the malware into finite sets of API calls. The set of high-level goals is referred to as the malware's behaviour (Martignoni et al., 2008).

Dynamic analysis of malware behaviour has been used to extract specifications of malware behaviour (Christodorescu et al., 2008). The specifications are derived from API call profiles obtained from dynamic analysis. These malware specifications are used for the detection of malware. Malware capabilities labelling, using Linear Temporal Predicate Logic (LTPL) operating on a dynamic trace of file system access, has been used to identify malware capabilities as part of a malware detection system (Mankin, 2013).

Singh (2002) proposes the modelling of the malicious behaviours of malware by using tuples of Subject, Object, Action, and Function, where subject represents active system entities, object is a system resource which is used to store information, action is the malicious behaviour which is initiated by a trigger, and function is the outcome of the action performed by a subject on an object.

Grégio et al. (2015) propose a set of malware behaviours in order to build an extensible malware taxonomy. These behaviours are grouped into classes. The classes are, Evasion, Disruption, Modification, and Stealing. The Evasion class contains behaviours pertaining to the removal of evidence, removal of registries, anti-verus engine termination, and firewall termination. The Disruption class contains behaviours for the scanning of known vulnerable services, email sending, IRC/IM known port connection, and IRC/IM unencrypted commands. The Modification class contains behaviours pertaining the creation of a new binary, creation of unusual mutex, modification of name resolution file, modification of browser proxy settings, modification of browser behaviour, persistence, download of known malware, download of unknown files, and driver loading. The Stealing class contains behaviours pertaining to the theft of system/user data, theft of credentials or financial data, and process hijacking (Grégio et al., 2015). It is noted that the behaviours identified by Grégio et al. are not complete, for example, gaining control of the command line or desktop (Backconnect) could be considered to be a member of the Modification class.

3. Malware behaviours

The MAEC language is provided to permit the sharing of structured information about malware (Kirillov et al., 2014b). The MAEC language defines low level actions, mid-level behaviours, and high-level capabilities in relation to malware. Low level actions are those actions that can be performed by malware through OS API calls. Mid-level behaviours aim to organise and describe the intention behind low level actions. Behaviours describe the operation of a malware instance at an abstract level and consist of groups of low level actions (Lee et al., 2013). Capabilities provide a description of the full range of behaviours of malware. MAEC provides a non-exhaustive list of malware capabilities which are shown in Table 2 (Kirillov et al., 2014a). The MAEC language not only provides a language for exchanging information about malware, it formalises the concepts of malware low-level actions, behaviours, and capabilities. Some of the default MAEC behaviours are broad in scope and may be difficult to map into low level actions, e.g the Fraud behaviour.

Banking malware employs the following key capabilities: command and control, data theft, spying, integrity violation, data exfiltration, fraud, and persistence. This paper uses the

Table 2 – MAEC capabilities and the behaviours used in this paper.

MAEC Capability	Behaviour Name
Command and Control	Configuration
Remote Machine manipulation	
Privilege escalation	
Data theft	Info Stealing, Injection
Spying	Screenshot, Video Capture
Secondary Operation	
Anti-detection	Anti-Analysis
Anti-code analysis	Anti-Analysis
Infection/Propagation	
Anti-behavioural analysis	Anti Analysis
Integrity violation	Process Injection
Data Exfiltration	Network Communications
Probing	
Anti-removal	Persistence
Security degradation	Info Stealing, Injection
Availability violation	
Destruction	
Fraud	Configuration, Info Stealing, Injection
Persistence	Persistence
Machine access/control	Backconnect, Network Communications

following non-exhaustive set of behaviours: Persistence, Configuration, Process Injection, Information Stealing and Injection, Network Communications, Backconnect, Screenshot and Video Capture, and Anti-Analysis. These behaviours were identified as they represent the core behaviours of the selected banking malware families. A comparison of the implementation of these behaviours is used to demonstrate the degree of similarity between banking malware families. A comparison of MAEC capabilities to the behaviours used in this paper is shown in Table 2. As the MAEC capabilities represent all malware families, they are a superset of banking malware behaviours.

The Application Program Interface (API) calls referred to in this paper are assumed to be Windows OS API calls, the details of which are provided in Microsoft developer documentation (Microsoft, 2016).

The remainder of this section discusses how each of the key malware behaviours might be implemented in the case of the Windows OS. The following section will then review the implementation used by each malware in our selection.

3.1. *Persistence*

Modern OS's have a function to automatically start programs, including malware. In the Windows OS, these are known as Auto-Start Extensibility Points (ASEP) (Blunden, 2012; Wang et al., 2005). When malware is installed, it uses the OS ASEP capability to ensure that it is started and to ensure its persistence (Mankin, 2013). The `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` registry entry is often used for malware start-up. The registry entry is commonly referred to as the registry run key or the run key (Carvey, 2005; Sikorski and Honig, 2012; Wang et al., 2005).

The `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs` registry entry contains a list of Dynamic Link Libraries (DLLs) loaded into every GUI process when it is started (Blunden, 2012; Carvey, 2009; Wang et al., 2004, 2005). The winlogon program creates events for actions such as logon, logoff, start-up, shutdown, and lock screen (Blunden, 2012). The `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon` registry entry contains the name of the DLL that will be called to handle a specific Windows event. This registry entry may be used for malware purposes.

The Windows OS runs system tasks as services, which are DLL files that are run by the svchost program. It is common to see several svchost instances running on a Windows OS. The regsvr32 program is used to register DLL files as Windows services. Malware DLLs are commonly run as a service using the regsvr32 command or by adding an entry to `HKLM\SYSTEM\CurrentControlSet\Services` (Carvey, 2005; Sikorski and Honig, 2012; Wang et al., 2005). A malware installer may patch (trojanize) an OS program. The patch inserts malware code and then returns control to the OS program (Sikorski and Honig, 2012). There are many other registry locations that are used for malware start-up. The Autoruns program displays the programs that automatically run when the Windows OS is started (Sikorski and Honig, 2012). A malware program can protect its persistence with the use of a thread or a process that is used to recreate the persistence mechanism when it is removed from the OS start-up locations (Malware Digger, 2015a).

The Carberp and Rovnix malware families make use of a bootkit (contraction of "boot loader" and "rootkit"). The bootkit allows malware code to load an unsigned driver and maintain control through the boot process. Bootkit techniques allow the malware developer to circumvent the Windows signed driver policy and to maintain persistence when the OS is re-installed (Rodionov et al., 2014). The malware driver may be used to create a hidden, encrypted Virtual Filesystem (VFS). I/O Request Packet (IRP) filtering is used by the malware driver to prevent the encrypted filesystem from being read or overwritten. The malware driver is used to inject malware code from the hidden filesystem into a target process (Rodionov et al., 2014; Rodionov and Matrosov, 2012).

3.2. *Configuration*

Banking malware uses a configuration (Kolbitsch et al., 2010) that provides details of the financial institutions that are being targeted, data to be displayed to the victims, and it may contain scripts to automatically transfer funds from the victim's account. The configuration may contain webinjects data or include URL links to webinjects scripts. A webinject consists of HTML/Javascript which is injected into a webpage. Webinjects may be present in the configuration or the configuration may link URLs where the webinjects can be downloaded. Webinjects are used to defeat the security measures of a specific bank. Webinjects may be used for a variety of purposes including automatic funds transfer (Binsalleeh et al., 2010; Boutin, 2014; Forrester, 2014; Kharouni, 2012). The configuration stored on the victim's computer is encrypted to avoid detection. The encryption used by the banking malware includes AES, TwoFish, RC2, RC4, RC6 and custom encryption. Custom encryption functions are developed by malware authors using bit and arithmetic operators. Custom encryption systems do not provide strong encryption, however they are novel and may cause existing analysis tools to fail (Kroustek, 2015; Malware Digger, 2015b; Matrosov et al., 2011; Milletary, 2012; Rodionov and Matrosov, 2012; Rossow et al., 2012; Wyke, 2011).

Banking malware configuration is generally implemented as a static configuration within the malware sample, while a dynamic configuration is downloaded from the attacker's Command and Control (C2) server. The static configuration contains basic details such as the IP addresses or hostnames of the C2 server, a customer identifier and possibly a cipher or encryption key (Binsalleeh et al., 2010; Wyke, 2015). The dynamic configuration may contain details of websites to ignore, websites to target and the actions to be performed when a specified website is visited. The use of a dynamic configuration gives the attacker flexibility in controlling the attack, thereby allowing configuration changes in response to bank countermeasures and changing targeting requirements (Al-Bataineh and White, 2012; Binsalleeh et al., 2010).

A criminal group running a large network of compromised computers (a botnet) may partition the botnet so that their criminal "customers" have access to data stolen from a specified section of the botnet. This is performed by embedding a customer ID into the configuration of the banking malware.

3.3. *Process injection*

Process injection is a technique commonly used by malware in order to hide the malware code by running it within a benign system process. Due to the complexity of process injection, the method used by a malware family may be considered to be invariant (Barabosch et al., 2014). Process injection allows the injected malware to utilise the access level of the target program. Malware code can be injected into a web browser process in order to obtain internet access (Sikorski and Honig, 2012). Malware process injection can be performed using a number of techniques including DLL injection, direct injection, process hollowing, and Asynchronous Procedure Call (APC) injection.

DLL injection is a common method to cause a process to execute malicious code. DLL injection operates by injecting code into a remote process. When this code is executed, it calls the `LoadLibrary` API with the name of the malicious DLL. The OS then loads the malicious DLL and begins execution of the `dllmain` function. The Windows APIs most often used in DLL injection are: `VirtualAllocEx`, `WriteProcessMemory`, `GetModuleHandle`, `GetProcAddress`, and `CreateRemoteThread` (Berdajs and Bosnić, 2010; Kasslin et al., 2005; Sikorski and Honig, 2012; Sun et al., 2006).

Direct injection uses APIs similar to those in DLL injection, except it is not necessary to force the target process to load a malicious DLL. The Windows APIs commonly used in direct injection are: `OpenProcess`, `VirtualAlloc`, `OpenProcess`, `WriteProcessMemory` and `CreateRemoteThread`. Calls are made to `VirtualAllocEx` and `WriteProcessMemory` in order to copy the data and code of the malicious program into the target process. When the injected code begins execution, it obtains the addresses of the required APIs using `LoadLibrary` and `GetProcAddress` calls or by using custom functions. Direct Injection requires a higher skill level on the part of the malware author (Sikorski and Honig, 2012; Sun et al., 2006).

Process hollowing starts by creating a benign process in a suspended state. `GetThreadContext` is used to obtain a thread in the process and `ReadProcessMemory` is used to read the memory of the process. `ZwUnmapViewOfSection` is used to deallocate the memory associated with each of the program sections. `VirtualAllocEx` is used to allocate memory and malware code is copied into the newly allocated memory using the `WriteProcessMemory.` `SetThreadContext` and `ResumeThread` are used to start process execution. An anti-virus program running on the victim's machine will see the name and PE header of the benign process and may not detect the executing malicious code (Sikorski and Honig, 2012).

The APC facility, provided by the Windows OS allows a function to be called in response to an asynchronous event. Normal execution of the program is interrupted and the called function executes as a thread in its own context. APC's may be issued for kernel mode or for user mode code (Alexander, 2012; Sikorski and Honig, 2012). The PowerLoader injection method is an example of APC injection that is used to target the Windows explorer process. In this technique, a pointer in the taskbar's window handling procedure is overwritten. A handle for the taskbar is obtained using the `FindWindow` API. The function pointer for the taskbar's window handling procedure is located at the start of the extra window memory. Extra window memory is accessed using the `GetWindowLong` and `SetWindowLong` APIs. A `GetWindowLong` call is made to obtain the address of the taskbar's window handler function. A `SetWindowLong` call is made to set the handler function pointer to the injected malware function. Finally, a call is made to the `SendNotifyMessage` API to cause execution of the malware function (Author Unknown, 2013; Dietrich, 2014).

3.4. *Information stealing and injection*

A key characteristic of banking malware is to lower OS integrity in order to steal user data and to modify data presented to the user. The lowering of OS integrity is performed by hooking API calls to alter flow control (Garcia-Cervigon and Llinas, 2012). The hooking of API calls can be performed in several places in non-privileged (user-mode) and privileged execution (kernel mode). User-mode hooking techniques include the hooking of the Import Address Table (IAT) and inline hooking. User-mode hooking is performed on a per process basis. However, system mode hooks are applied to all processes. System mode hooking is more demanding because it is not well documented and errors in the hooking software can cause system failure (Kasslin et al., 2005; Mankin, 2013). System mode hooking techniques include the hooking of the System Service Dispatch Table (SSDT), API call patching, Direct Kernel Object Modification (DKOM) and the addition of a malware file system driver.

There is one IAT per process and each IAT contains pointers to functions contained in the DLLs that are used by the process. The IAT is populated by the OS loader. In IAT hooking, the malware first saves the original API call function pointer and replaces it with a call to a malware function. Before the malware function exits, the original API call is made (Berdajs and Bosnić, 2010).

In inline hooking, the first five bytes of an API call are saved and then overwritten to jump to a malware function. The saved bytes are then executed at the end of the malware function in order to call the API call. As the Intel 32 bit architecture uses variable length instructions, a disassembler function can be used in the hooking process in order to determine instruction lengths to ensure that overwriting does not corrupt instructions and cause a system crash. The inline hooking approach is also used by the Microsoft Detours library (Berdajs and Bosnić, 2010; Hunt and Brubacher, 1999; Tsaur and Chen, 2010).

When an API call is made by a user-mode program, the API call provides an interface to the underlying system code operating in privileged mode. An API call dispatch ID is passed to the SYSENTER instruction. The SYSENTER instruction is used to enter ring 0 or privileged mode execution. The API call dispatch ID is used as an index to the SSDT to obtain a function pointer for the required API call. Overwriting of a function pointer in the SSDT allows hooking of an API call. Unlike non-privileged hooking methods, SSDT hooking provides hooking of an API call for all processes (Kasslin et al., 2005; Mankin, 2013).

System code may be patched and patching can be performed on disk or in memory. The patching of the system code requires a higher skill level and is harder to detect than user-mode hooking. Patching can be used to disable tests, provide inline malware code, or to jump to malware code in a separate memory allocation (Mankin, 2013).

DKOM provides methods to access and modify kernel data structures relating to processes and drivers, in order to hide processes or drivers. Windows processes and drivers are represented as data structures on double linked lists. A process may be hidden by removing the corresponding data structure from the process list. One difficulty with the DKOM approach is that much of the Windows OS kernel internals are not documented and may be changed by patches or by the release of OS updates (Kasslin et al., 2005; Mankin, 2013).

Windows provides a layered device driver architecture. In this architecture, IRPs that represent driver requests are passed between the objects at different layers of the device driver stack. The insertion of a malware device driver can be used to log keystrokes or to hide directories and files by dropping IRPs (Mankin, 2013).

The research by Liang et al. (2008) resulted in the HookFinder program. This program uses dynamic analysis to identify hooking behaviours in malware samples without prior knowledge of hooking mechanisms.

3.5. *Network communications*

Banking malware uses encryption of network traffic in order to avoid detection by intrusion detection systems (IDS) and intrusion prevention systems (IPS). TLS/SSL, RC4, AES, and custom encryption schemes are commonly used.

Stolen victim credentials and malware statistics are valuable information for the attacker. These details are passed back to a C2 server controlled by the attacker. This process is known as exfiltration (Al-Bataineh and White, 2012). Custom protocols are generally used by malware for loading a dynamic configuration or exfiltrating stolen data. These protocols frequently use common TCP/IP ports (HTTP, HTTPS and DNS) as firewalls are often configured to pass data that uses these ports (Born, 2010).

Banking malware can use a Domain Generation Algorithm (DGA). A DGA is an algorithm that produces domain names that may be based on date. The generated domain names are registered in advance by the attacker. The generated domain names are used as a fallback measure in cases where hardcoded C2 servers are not available (Kolbitsch et al., 2010). A DGA can generate a large number of domain names. This may also be an attempt to hinder analysis (Antonakakis et al., 2012). Replication of date based DGA algorithms can be used to identify malware related domains. DGA domain names generated by replication may be used to identify malware instances and may be used in remediation (Plohmann et al., 2016).

URL redirection can be used to prevent the victim from accessing an anti-virus vendor's website. It can also be used to redirect the victim to a phishing website. URL redirection may be performed by adding extra entries to the hosts file, by setting up a proxy autoconfiguration file, or by hooking API calls (Grégio et al., 2015).

3.6. *Backconnect*

Backconnect provides an attacker with access to the command line or desktop of an infected computer. Desktop access on the victim's computer allows banking transactions to be made by the attacker while the victim is connected to an internet banking session. A backconnect session is initiated when a command is sent from the attacker's C2 server to the victim's computer. Malware installed on the victim's computer responds by opening a connection to the attacker's C2 server. Initiating the connection from the victim's computer to the attacker's C2 server bypasses connection problems due to firewall or Network Address Translation (NAT) connections. A backconnect facility can be implemented using SOCKS, FTP, HTTP, Virtual Network Computing (VNC) or Remote Desktop Protocol (RDP) protocols (Dos Santos, 2012; Sood and Enbody, 2014). An advantage of using the VNC protocol is that VNC allows the victim and attacker to be simultaneously connected to a compromised computer. If an attacker using the RDP protocol connects to a compromised computer while the victim is using the computer, then the victim will be logged out (Cherepanrov and Lipovsky, 2013).

A SOCKS or HTTP proxy provides the attacker with the ability to login to an internet banking website which, through the lens of bank security software, appears as though the banking session is originating from the IP address of the victim's computer.

3.7. *Screenshot and video capture*

The screenshot behaviour allows an attacker to capture images of the desktop, showing details such as the use of a virtual keyboard to enter credentials for a banking session. Video capture allows recording of a video of the users banking session, providing the attacker with a better understanding of the victim's banking session (Milletary, 2012).

3.8. *Anti-analysis*

Strings in malware are commonly encrypted in order to delay analysis and prevent simple techniques being used to identify the malware. Simple custom encryption methods using bit or arithmetic operations, base64 encoding, and substitution ciphers are commonly used for string obfuscation (Cannell, 2015; Giuliani and Allievi, 2010; Kroustek, 2015; Sikorski and Honig, 2012; Trend Micro, 2010). Strings in malware may also be encrypted with stronger encryption algorithms (Aronov, 2015).

API calls are commonly obfuscated in malware in order to delay analysis and to prevent simple methods from being used to understand the malware. Commonly used API call obfuscations include the use of an API call handler function that may be passed integer or hash values representing the API to be called. Another technique is to copy API code to a new memory allocation within the malware program (Cherepanrov and Lipovsky, 2013; Giuliani and Allievi, 2010; Suenaga, 2009).

There are a number of well-known virtual machine (VM) detection methods. These methods include Microsoft Virtual PC detection using custom instruction codes, VMWare detection using the control port, Interrupt Descriptor Table (IDT) register testing, checking of system services, checking the MAC address of the network card, checking for VM specific hardware devices, filesystem checking, use of the `cpuid` instruction to test for hypervisor execution, checking for VM specific registry keys and testing the number of CPU cores (Brand, 2010; Kovacs, 2015; Lau and Svajcer, 2010). The Rovnix malware creates a system crash by sending an IoControlCode to a Sysinternals driver that is installed as part of the Rovnix installation. The aim of this is to halt automated analysis in a VM (Malware Digger, 2015a).

Malware anti-analysis features may prevent installed malware samples from being collected and then executed on a computer different from the installation computer. These techniques operate by generating data from a feature of the installation computer and patching the installed malware program with this data. When the installed program is executed, the feature data is checked against the current execution environment. An example of this is the use of the hard disk volume identifier by Zeus malware (Wyke, 2011). When the Citadel malware detects that it is executing in a VM, it attempts to connect to a randomly generated C2 address. This is an attempt to mislead analysis into concluding that the sample is not active (Milletary, 2012; Rahimian et al., 2014).

4. Malware families

This section gives details of the following banking malware families: Zeus and the Citadel variant, Vawtrak, Dridex, Dyre, Carberp and Rovnix. Descriptions of the operation of the above malware families are based on research papers, industry reports, and web pages. These sources describe potentially different versions of malware and, in some instances, present conflicting details of malware operation.

4.1. *Zeus malware*

The first version of the Zeus malware (Zeus V1) also known as Zbot was detected in 2006 (Ligh and S. S. Corporation, 2006; Riccardi et al., 2011). Zeus was the first banking malware kit. It was sold publicly and little technical knowledge was required to create a Zeus botnet. Zeus version two (Zeus V2) was first detected in 2010. Zeus V2 provided the facility to have multiple instances of Zeus running on one computer and the use of RC4 encryption in Electronic Code Book (ECB) mode (Selvaraj, 2010). Zeus V2 malware has the following capabilities:

- Rule based stealing of user data from HTML forms,
- Rule based injection of data into a websites HTML,
- URL redirection to websites controlled by the attacker,
- Capture of the HTML of targeted websites,
- Stealing of cookies,
- Deletion of cookies,
- Stealing of mail and FTP account credentials,
- Download and execution of additional programs and
- Built-in virtual network computing (VNC) console.

Zeus malware achieves persistence by adding the Zeus program to the `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` registry entry (Wyke, 2011).

The Zeus V2 program contains a `xor` encrypted static configuration that includes a dynamic-configuration-URL. The dynamic configuration includes the URL where stolen data can be uploaded, trigger conditions for screenshots, the URL where the updated versions of the malware can be downloaded, URL-masks to control logging, URL redirection pairs, IP and hostname pairs for URL redirection, and webinjects. When the dynamic configuration is decrypted, a recursive `xor` procedure is performed to decode the decrypted data. This procedure is known as "visual decrypt". The decoded data is then decompressed using UCL (NRV2B) decompression (Baumhof and Shipp, 2011; Wyke, 2011). The encrypted Zeus configuration data is stored in a randomly named registry key (Alvarez, 2013; Wyke, 2011). Stolen data is encrypted using a second RC4 key. This data is temporarily stored into a file before being sent to the C2 server. Zeus V2 malware uses three threads: a thread for receiving network data, a thread that downloads the configuration data, and a thread that checks the malware's start-up key in the Windows registry and replaces it if it is removed (Wyke, 2011).

Zeus uses a direct injection method to inject its malware code into other processes. Zeus iterates through processes using the `CreateToolhelp32Snapshot`, `Process32FirstW`, and `Process32NextW` APIs. The system and the Zeus processes are skipped. A mutex is then created using a `CreateMutexW` call. Each selected process is opened with an `OpenProcess` call. An access token for this process is obtained by an `OpenProcessToken` call with a requested access of TOKEN_QUERY. If the access token is not available, iteration continues with the next process. The process is then opened using an `OpenProcess` call. The access mode specified on the `OpenProcess` call checks for full access to the target process. If the current process is accessible, then memory is allocated in the target process using a `VirtualAllocEx` call. The Zeus malware is copied to the target process using a `WriteProcessMemory` call. A handle to the previously created mutex is passed using a `DuplicateHandle` call. The injected Zeus code is then started using a `WriteProcessMemory` call. The current process and the generated mutex are then closed by `CloseHandle` calls (Alvarez, 2013).

When a Backconnect command is sent from the attacker to the compromised computer, a network connection is opened from the victim's computer to a server controlled by the attacker. This allows network connections to a firewalled machine (Falliere and Chien, 2009). The Backconnect facility provides command line access and allows the use of the RDP, Socks, and FTP protocols (Falliere and Chien, 2009; Zeus Author, 2011).

Zeus contains a Virtual Network Computing (VNC) based facility that allows the attacker to connect to the Graphical User Interface (GUI) of the victim's computer and perform banking transactions from the victim's IP address and allows access to hardware based authentication mechanisms used by the victim (Stevens and Jackson, 2010).

Zeus contains a screenshot facility. The trigger conditions to take screenshots are provided in the configuration. An example would be taking a screenshot of the banking balances of an internet banking session. The trigger condition would be the URL used by a specific bank to display balance information. Screenshots can be taken by a left mouse click on the virtual keyboards used by banking websites (Ligh and S. S. Corporation, 2006; Stevens and Jackson, 2010).

Zeus provides a URL redirection facility that is used to take the victim from the desired website to an attacker controlled website (Stevens and Jackson, 2010).

An anti-analysis feature of the Zeus malware involves creating a GUID from the volume identifier of the hard disk on which the malware is installed. This GUID data is then written into the installed malware program. When the Zeus malware is executed, an action is taken to check the internal GUID data against a GUID created from the volume identifier of the hard disk from the execution environment. If the two GUIDs do not match, the malware terminates (Wyke, 2011).

Zeus uses inline hooking of API calls (Ligh and S. S. Corporation, 2006). Zeus V1 hooks the `NtQueryDirectoryFile` API call to hide its files and uses a simple fixed key encryption algorithm (Wyke, 2011). Zeus V2 does not hide its files (Wyke, 2011). A weakness with Zeus V2 is that its configuration data could be directly downloaded by security researchers and decrypted using the configuration RC4 key contained within the malware sample.

The source code for the Zeus V2 malware was leaked online in April 2011 (Baumhof and Shipp, 2011; Rahimian et al., 2014; Riccardi et al., 2011). The public availability of the Zeus source code led to the development of a number of Zeus variants including Citadel and ICE-IX. Another Zeus variant was the Zeus Peer to Peer (Zeus P2P) malware, also known as Gameover Zeus (GOZ).

4.2. *Citadel malware*

The Citadel malware was derived from the leaked Zeus V2 source code. Citadel was first detected in January 2012 (Milletary, 2012). Citadel is based on the Zeus source code, so there are significant similarities between the Zeus malware and the Citadel malware. The facilities of the Citadel malware are similar to those of the Zeus malware with the addition of the following improvements:

- Modified RC4 algorithm,
- Video capture,
- Sandbox detection,
- Support for Google Chrome,
- Securing of the configuration file,
- AES cryptography,
- VM detection,
- Automated DOS command line injection and
- Distributed denial of service attack capability (Milletary, 2012; Rahimian et al., 2014).

Citadel maintains persistence by adding the Citadel program to the `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` registry entry (Rahimian et al., 2014). Citadel configuration data is stored in the registry in the same manner as Zeus. Citadel samples contain a 32 byte "hex as ASCII" value that is referred to as the login key. In Citadel version 1.3.4.5 the RC4 algorithm was modified by the addition of `xor` operations (AhnLab, 2012; Milletary, 2012; Rahimian et al., 2014). Citadel uses AES 128 bit encryption in ECB mode (Milletary, 2012). The AES encryption key is created by taking an MD5 hash of the login key and then using an RC4 key to encrypt this MD5 hash (Milletary, 2012). Version 1.3.4.5 uses a modified RC4 algorithm. The modification consists of adding extra `xor` operations to the RC4 encryption function. In version 1.3.5.1, the AES function was modified by the use of an additional 128 bit `xor` key. This key is used to `xor` each block of data prior to AES encryption or after AES decryption. Citadel uses UCL compression as does Zeus. Citadel communicates with the C2 server using the HTTP protocol with RC4 encrypted data (Milletary, 2012).

The security of the Citadel configuration was improved by requiring an encrypted request for the configuration. To create a request to download a Citadel configuration, a new RC4 key is generated using a modified RC4_init function and static values embedded within the Citadel program. The new RC4 key is used to encrypt a configuration request data structure prior to sending to the Citadel C2 server (Rahimian et al., 2014; Wyke, 2012). Citadel uses the same direct process injection technique as Zeus (Barabosch et al., 2014).

Citadel uses the same backconnect techniques as Zeus. Citadel contains a screenshot facility and has a video capture plugin to provide a live video recording of operations on the victim's screen. Citadel contains a VNC server and a SOCKS server (Milletary, 2012; Sood and Rohit, 2014).

Citadel performs VM detection by checking strings in the resources section of each process running on the system. When a VM is detected, Citadel attempts to connect to a randomly generated URL. This is an attempt to deceive analysis and to indicate that the sample is no longer active (Milletary, 2012; Rahimian et al., 2014). The Citadel malware contains a list of anti-virus websites which it prevents to by using DNS redirection (Rahimian et al., 2014).

4.3. *Vawtrak malware*

Vawtrak malware belongs to the Gozi malware family and is also known as Neverquest or Snifula (Kimberly, 2014b; Kroustek, 2015; Wyke, 2014). Gozi malware was first detected in 2007 (Jackson, 2007). Gozi was a competitor to Zeus malware and used a “Crimeware As A Service” business model (Wyke, 2014). Vawtrak malware was first detected in August 2013 (Trend Micro, 2015). A new version of the Vawtrak malware was detected in October 2015 (Huss and Matthew, 2015). The following description refers to the Vawtrak variant from 2013. Vawtrak has the following facilities:

- Keystroke logging and credential stealing (Kroustek, 2015),
- VNC server, to allow remote control of the victim’s computer (Kroustek, 2015; Wyke, 2014),
- SOCKS server, to allow the attacker to utilize communications with the IP address of the victim’s computer (Kroustek, 2015; Wyke, 2014),
- Screenshot and video capture facilities (Kroustek, 2015; Wyke, 2014),
- A debugging feature that may also be used by malware analysts (Wyke, 2014),
- Ability to hinder anti-virus software by setting a Software Restrictions Policy (SRP) in the Windows registry. This SRP setting restricts the privileges of anti-virus processes (Kroustek, 2015; Wyke, 2014),
- Ability to disable the IBM Trusteer Rapport security program by hooking the VirtualProtect API (Kroustek, 2015),
- Support for information stealing and injection in Internet Explorer, Firefox and Chrome web browsers (Kroustek, 2015).

Vawtrak uses the `RegCreateKey` and `RegSetValueEx` API calls to create `HKCU\Software\Microsoft\Windows\CurrentVersion\Run` registry entry to automatically start the Vawtrak malware DLL using the regsvr32 program (Alvarez, 2015; Kroustek, 2015; Wyke, 2014).

Vawtrak uses an RNG function that is similar to the Linear Congruential Generator (LCG) employed in Visual C++ (Wyke, 2014). The LCG seed is stored in the header of encrypted data. The stream of pseudorandom numbers generated by the LCG function are used to perform `xor` encryption.

Communications with the C2 server use the HTTP protocol (Kroustek, 2015). Some Vawtrak C2 servers are accessed using The Onion Router (TOR) protocol. TOR C2 hostnames are generated using a DGA with hardcoded seed values (Alvarez, 2015). Data sent to the C2 server is obfuscated by a custom encoding function (Kilman, 2014), and in later versions, this data is also base64 encoded. Vawtrak configuration data from the C2 server is encrypted using LCG based encryption and is compressed using aPLib compression (Asinovsky, 2014; Ibsen, 2014; Kroustek, 2015). Initial C2 addresses are contained in the embedded static configuration. When an updated C2 server list is downloaded, signing is verified with an embedded RSA public key (Kroustek, 2015). Encrypted and compressed configuration data is stored in a randomly named registry key (Kroustek, 2015; Wyke, 2014).

The Vawtrak configuration contains a list of target URLs and the associated webinjects. Following this is a list of social networking, entertainment, and shopping website URLs. When these URLs are encountered, credentials are extracted and sent back to the C2 server. These credentials are used to further spread the Vawtrak malware. Following the URL list is a list of banking related phrases. When these phrases are detected, the contents of the webpage are sent to the C2 server and are used to facilitate the development of webinject scripts to target additional organisations (Wyke, 2014). The Vawtrak configuration includes a Project ID that is used to partition the botnet and is specific to each Vawtrak customer (Kroustek, 2015).

On a 32 bit OS, Vawtrak injects its malware into processes using direct injection (Kroustek, 2015). On a 64 bit OS, APC injection (Wyke, 2014) using the PowerLoader technique is used to target the explorer process (Kimberly, 2014b). Vawtrak malware is injected into all processes at the security level of the injecting process. The injected code terminates if it is running in a system process (Kroustek, 2015).

Vawtrak uses inline hooking of user-mode API calls and contains a basic disassembler which is used to check the length of instructions prior to overwriting (Kroustek, 2015).

Vawtrak contains a VNC capability that provides the attacker with access to the desktop of the compromised computer. Vawtrak supports communications from the network address of the compromised computer using a built in SOCKS server (Kroustek, 2015; Wyke, 2014).

4.4. *Dridex malware*

Dridex malware was first detected in 2014 (Certeza, 2015; Sanghavi, 2016). Dridex is also known as Bugat or Geodo (abuse.ch, 2014; Kimberly, 2014a; Thadani, 2016). Dridex is a variant of Cridex (abuse.ch, 2014). Cridex was first detected in 2012 and included a self-propagation capability which is not present in the Dridex variant (O’Brien, 2014). In order to make the botnet more resistant to takedown efforts Dridex implemented a P2P communications topology in November 2014. Dridex communications are multi-layered. The outermost layer is composed of compromised computers. Compromised computers which are not behind NAT firewalls may be used as peer nodes which form the second communications layer. The third layer is a frontend which handles communications with the C2 computers which form the fourth layer (Blueliv, 2015; O’Brien, 2014). Unlike malware which is sold as crimeware kits, Dridex is a single botnet which is partitioned into subnets which divide the botnet based on attacker requirements. Dridex subnets are identified by numeric identifiers e.g. botnet 120, botnet 220 (O’Brien, 2014).

Dridex uses the following modules:

- Loader module,
- Main module,
- SOCKS module,
- VNC module,
- MOD4 server module,
- MOD6 server module,

The Dridex loader module is responsible for downloading the main Dridex module. The loader module contains a static configuration which includes the IP address to download the main module and the malware configuration. These items are requested using a RC4 encrypted XML message (O'Brien, 2014).

The main module provides the following functions:

- Backconnect,
- Cookie stealing,
- Credential theft,
- File operations,
- Command line processing,
- Information stealing and injection,
- Keystroke logging,
- Peer node server,
- Stealing of user information from HTML forms,
- Screenshots,
- HTTP redirection,
- Attacks Chrome, Firefox and Internet Explorer web browsers,

A remote graphical interface to the compromised computer is provided by the VNC module. Command line execution and file system operations are provided by the SOCKS module. Execution of new processes is provided by the mod4 module. Spamming using the contact list of the compromised computer is provided by the mod6 module. Dridex provides 32 bit and 64 bit modules (O'Brien, 2014).

Dridex is started by the creation of a registry run key (Blueliv, 2015). Dridex uses DLL injection to inject the main module into the explorer process (Rocha, 2016). When the Dridex loader runs, it deletes the Dridex run key from the registry and deletes the Dridex loader from the file system. The Dridex run key and loader are re-written on system shutdown (Kimberly, 2014a; Rocha, 2016). Dridex uses obfuscation of strings and API calls. Dridex uses an encrypted XML based protocol which operates underneath TLS communications. Dridex contains a hardcoded copy of the C2 public key. At initialization, Dridex generates a new RSA keypair, the new public key is encrypted with the C2s public key and sent to hardcoded layer 3 server. The Dridex modules and a node list are then requested. All subsequent communications are made through the peer nodes. The Dridex configuration is then requested. The Dridex protocol uses RC4 and RSA encryption. An RC4 key is encrypted using the generated RSA private key, the remainder of the message is RC4 encrypted. Dridex uses `xor` based custom encryption in the protocol for downloading modules. Binary data in the protocol is base64 encoded. Following module download, Dridex communications are directed to the Dridex nodes (Blueliv, 2015; O'Brien, 2014). The Dridex configuration is stored in a randomly named registry key using custom encryption and aPLib compression (Su, 2015).

4.5. *Dyre malware*

Dyre malware was first detected in 2014 (Shulman and Dorfman, 2015). Dyre is also known as Dyreza, Dyzap or Dyranges (Stone-Gross and Khandhar, 2014). Dyre malware has the following capabilities:

- Man in the middle attack,
- DGA,
- Invisible Internet Project (I2P) tunnelling support,
- Built-in VNC server,
- 32 bit and 64 bit payloads (Kimberley, 2014b; Stone-Gross and Khandhar, 2014),
- AES encryption,
- Single partitioned botnet,
- Attacks Chrome, Firefox and Internet Explorer (Kimberley, 2014b),

The first section of the injected Dyre payload contains position independent code that rebuilds the Import Address Table (IAT) (Chiu and Villegas, 2015). The Dyre installer writes a randomly named executable file to the "Application Data" directory (Hanel, 2014). A mutex is created to check whether the malware is already installed. Persistence is achieved by creating a run key in `HKCU\Software\Microsoft\CurrentVersion\Run` (Hanel, 2014; Kimberley, 2014b; Stone-Gross and Khandhar, 2014). A call is made to `IsWow64Process` to determine whether it is running on a 64 bit system. The Dyre Installer contains resources incorporating 32 bit and 64 bit versions of the Dyre payload (Stone-Gross and Khandhar, 2014). The Dyre installer runs and creates a service named "Google Update Service", which injects the Dyre payload into the target process, and then terminates (Chiu and Villegas, 2015; Kuhn et al., 2015; Symantec Security Response, 2015).

The `NtMapViewofSection`, `VirtualAloc`, `NtQuery SystemInformation`, `OpenThread` and `NtQueueApc Thread` API calls are used to perform APC injection of the Dyre payload into an svchost or explorer process (Chiu and Villegas, 2015; Hanel, 2014; Stone-Gross and Khandhar, 2014; Symantec Security Response, 2015). Dyre uses inline hooking of system API calls (Hanel, 2014). Dyre contains a table consisting of the compilation timestamps for the versions of the wininet DLL and the corresponding hooking offset. This table is used to identify the wininet version prior to patching. Unknown copies of wininet.dll are sent back to the C2 server (Hanel, 2014).

The static configuration is stored in a randomly named program resource. The Dyre payload reads the configuration data and writes an encrypted configuration to a file in the "Application Data" directory. The configuration is in an XML format that differs from the commonly used Zeus configuration format (Hanel, 2014).

If the infected computer uses a Network Address Translation (NAT) internet connection, then Dyre uses Session Traversal Utilities for NAT (STUN) to determine the IP address of the infected computer (Stone-Gross and Khandhar, 2014; Trend Micro, 2014). Dyre performs a man in the middle attack by redirecting website requests through an attacker controlled proxy server. Webinjects are added when the website response is received. Storing the webinjects on an attacker controlled server has the advantage of hindering efforts to access the webinjects for analysis (Kuhn et al., 2015).

Dyre connects to Invisible Internet Project (I2P) nodes to establish peer to peer tunnelling connections. When the network connections are established, Dyre uses the `CreateToolhelp32Snapshot`, `Process32FirstW` and `Process32NextW` APIs to locate the web browser processes. The Dyre malware DLL is injected into the Internet Explorer, Firefox and Chrome web browsers. Named pipes are used to pass commands and data between the Dyre malware and the Dyre DLL running in the web browser (Hanel, 2014).

If the Outlook email client is installed, then Outlook is hijacked to send emails containing the Dyre installer as attachments. The email recipients are obtained from the C2 server (Kuhn et al., 2015; Marcos, 2015).

Dyre has a modular architecture, a credential stealing module operates by capturing HTTP POST requests which are then sent to the C2 server. The stolen data is in a plain text format (Kimberley, 2014a; Shulman and Dorfman, 2015; Stone-Gross and Khandhar, 2014; Trend Micro, 2014). Dyre uses a VNC module (Kimberley, 2014a), which has three exported functions, these are ClientSetModule, VncStartServer and VncStop Server (Hanel, 2014). These functions are present in the VNC module in the Carberp source code. It is possible that leaked Carberp source code may have been used by the Dyre author to create the Dyre VNC facility (Hanel, 2014).

Incoming Dyre data is SSL encrypted. After SSL decryption, the data is AES encrypted. Configuration data and malware plugins are RSA signed (Stone-Gross and Khandhar, 2014; Trend Micro, 2014). Dyre generates a bot-id comprised of the computer name, the OS version and a 32 byte unique identifier (Kimberley, 2014a; Shulman and Dorfman, 2015).

Dyre malware contains a Domain Generation Algorithm (DGA) that uses the date as a key to generate the C2 server's IP address and port pairs (Chiu and Villegas, 2015). When the Dyre malware is initialised, the number of processor cores is checked. If this number is less than two, the Dyre malware terminates. This test is an anti-VM test to hinder security researchers (Lemos, 2015).

4.6. *Rovnix malware*

The Rovnix malware was first detected in 2011 (Harley, 2011). Rovnix makes use of a 64 bit bootkit that is similar to the bootkit used by the Carberp malware. The source code of the Carberp bootkit was leaked as part of the Carberp source code leak in 2013 (Kruse, 2013).

Rovnix achieves persistence by infecting the Volume Boot Record (VBR) of the active drive. When a computer running a Windows OS is started, the Basic Input/Output System (BIOS) reads the first sector of the bootable hard disk. This sector is known as the Master Boot Record (MBR). The MBR contains start-up code and a partition table. The MBR code scans the partition table and locates the partition with the bootable flag set. Then the code in the first sector (VBR) of the bootable partition is executed. File system specific partition start-up code is stored in the VBR. Rovnix malware infects the VBR of NTFS boot partitions. At installation, Rovnix compresses the original VBR code, replaces the original VBR with the Rovnix start-up code, and appends the compressed VBR start-up code. When the computer is started, the Rovnix start-up code obtains control. After the Rovnix start-up code has been executed, the original VBR code is decompressed and executed. The Rovnix start-up code in the VBR is polymorphic in order to avoid anti-virus detection (Carrier, 2005; Harley, 2011; Matrosov, 2012). In real mode, BIOS functions are accessed using the INT instruction. BIOS hard disk services are requested using INT 13h (Carrier, 2005; Scanlon, 1986). The Rovnix VBR start-up code performs hooking of the BIOS INT 13h handler. Hooking of the INT 13h allows patching of the ntloader/bootmgr, thereby enabling control to be maintained after the boot manager is loaded. When the boot manager is loaded, the original VBR code is decompressed and executed. During the system start-up process, the start-up code must switch from real mode to protected mode. In order to maintain control in the transition from real mode to protected mode, Rovnix makes use of the IDT. The IDT is a table that is used in protected mode and contains descriptors that provide access to interrupt and exception handlers (Blunden, 2012; Harley, 2011; Quist et al., 2006; US Air Force, 2000). The Rovnix malware copies itself into an unused section of the IDT. The malware then hooks the INT 1 protected mode debug handler (Blunden, 2012; Harley, 2011). It sets hardware breakpoints using the CPU's debugging registers dr0-dr7 in order to maintain control during Windows start-up (Brey, 2005; Harley, 2011).

The use of debugging registers allows the malware to gain control at specific points during the kernel loading process. This VBR infection technique is complex and requires a skilled developer. Rovnix malware operates on 32 bit and 64 bit versions of the Windows OS and allows an unsigned Rovnix driver to be loaded into the Windows OS (Harley, 2011).

Versions of Rovnix from 2012 used disk sectors near the end of the partition to create an RC6 encrypted Virtual File System (VFS). In this version, the configuration was stored in the VFS (Rodionov and Matrosov, 2012). A newer version of Rovnix from 2014 uses RC4 encryption and stores the VFS in a binary file (Feng, 2014).

A filtering driver is provided that prevents read or write access of the VFS file. If the Rovnix installer detects that full disk encryption software is installed on the victim's computer then the bootkit is not installed and the Rovnix banking malware is installed in the filesystem of the victim computer (Malware Digger, 2015a). The `NtCreateFile` and `NtDeleteFile` API calls are hooked to protect the VFS (Feng, 2014).

If the bootkit cannot be installed due to insufficient storage, it will install and run a copy of the Sysinternals Contig tool to defragment storage (Malware Digger, 2015a).

The Rovnix installer has the facility to either download the latest version of the Rovnix banking trojan or to install a copy of the Rovnix banking trojan from program resources. The Rovnix installer contains 32 bit and 64 bit versions of the Rovnix banking trojan. The stored copies of the banking trojan are compressed with a modified version of aPLib compression (Malware Digger, 2015a). The Rovnix trojan is initialised by a registry run key or by injection via the unsigned malware driver. If the Rovnix bootkit was not able to be installed, then a registry run key is created to start the Rovnix malware. The version of Rovnix banking trojan that is started from the filesystem contains a thread that protects the registry run key (Malware Digger, 2015b).

Rovnix uses a DGA to generate the C2 domain names. The DGA uses month as the key (Bitdefender, 2014). The Rovnix malware downloads additional plugins and external programs from the C2 server (Malware Digger, 2015b; Matrosov, 2012).

The plugins that are downloaded by Rovnix include a banking trojan known as ReactorDemo (Malware Digger, 2015b), a TOR client, a password stealer, and a bitcoin stealer. Rovnix plugins are identified by the Cyclic Redundancy Code (CRC32) value of the plugin. The downloaded plugins are RC2 encrypted DLLs. The decrypted data contains a RSA signature. The hashing algorithm used in this signature is not documented. Disassembly of a Rovnix sample showed the signature hashing algorithm to be SHA1. The TOR plugin contains a DGA that can be used as an alternative to TOR communications. The password stealer plugin may have been taken from the Zeus VM (KINS) partial source code leak (Malware Digger, 2015b).

The Rovnix banking trojan runs a server process on the infected computer. This process acts like a proxy server. The Rovnix banking trojan is injected into the Internet Explorer, Firefox, and Chrome browser processes. Target processes for injection are identified by comparing a CRC of the process name with a set of hardcoded CRC values within the Rovnix malware. The injected malware hooks a number of functions within the browser process. The hook routines perform communication with the Rovnix proxy server process. RC2 encrypted web-injects are downloaded from the C2 server. The proxy server process parses the webpages visited by the victim and performs HTML injection from the web-injects. The Symantec Anti-Virus company identifies Rovnix malware as a Carberp variant due to the shared bootkit and similar programming techniques (Malware Digger, 2015b).

4.7. *Carberp malware*

Carberp malware was first detected in June 2010 (Dolmans and Katz, 2013; Matrosov et al., 2011). The Carberp source code was leaked in June 2013 (Kruse, 2013). Carberp uses a bootkit that is similar to the Rovnix bootkit (Matrosov et al., 2011; Tigzy, 2013). It is possible that the same malware developer created both the Rovnix and Carberb bootkits (Matrosov et al., 2011). The bootkit allows the Carberp malware to persist after the reinstallation of the Windows OS (Tigzy, 2013). Carberp has the following facilities:

- Modular design, which is able to download and execute new modules from the C2 server. Carberp modules have an executable decryptor and are `xor` encrypted (Giuliani and Allievi, 2010),
- Screenshot module,
- Credential stealing module,
- VNC module to allow desktop access,
- A module to remove other banking malware,
- A module to disable anti-virus products,
- A module for generating traffic to perform Distributed Denial of Service (DDOS) attacks (Giuliani and Allievi, 2010; Kalnai, 2013),
- A capability to attack Internet Explorer and Firefox web browsers.

A Carberp variant was also produced that targeted the Android platform (Kalnai, 2013).

Carberp provides a general mode of attack which is independent of specific targeting. When a victim logs into an SSL website, Carberp intercepts the POST request and copies the victim's credentials. The stolen credentials are sent back to the C2 server. Carberp also supports a targeted attack when specific URLs are used to initiate information stealing (Trusteer, 2010). During the installation process, Carberp attempts to run privileged mode code to remove SSDT hooks from a number of operating system API calls. This is done to prevent anti-virus programs from interfering with the Carberp installation process (Giuliani and Allievi, 2010). The Carberp installer also attempts privilege escalation by exploiting several vulnerabilities (Matrosov et al., 2011).

A kernel mode driver is loaded by the Carberp bootkit. This driver injects a malicious DLL into the user-mode address space of processes running on the system. The bootkit is used in order to allow loading of an unsigned 64 bit driver on 64 bit Windows OS, thereby bypassing the driver signing policy of Windows 64 bit OS (Matrosov et al., 2011). The Carberp bootkit supports a hidden RC6 encrypted VFS that is allocated in unused storage. The VFS is a modified FAT16 file system. A filter driver is provided that prevents read or write access of the hidden virtual file system. The bootkit only supports NTFS partitions. The malicious DLL and other files are stored in the VFS. The malicious DLL is injected into the target process at system start-up (Maor, 2013). The Carberp bootkit generator uses metamorphism in the creation of the boot code. This provides a unique signature for the boot code for each Carberp malware user (Grill et al., 2014; Maor, 2013). Carberp uses function name hashing to obfuscate API calls (Giuliani and Allievi, 2010).

Carberp injects malware into every process and hooks the `NtQueryDirectoryFile` API call in order to hide the directory containing the Carberp program and configuration (Giuliani and Allievi, 2010). Carberp uses direct injection and APC injection. Carberp uses inline hooking of API calls (Trusteer, 2010).

Carberp uses the HTTP protocol for C2 server communications (Trusteer, 2010). There are two variants of Carpberp malware making use of RC2 or RC4 encryption. The variant using RC4 encryption was discontinued after an arrest in 2012 (Kalnai, 2013). The leaked Carberp source code uses RC2 encryption. Carberp also uses MD5 hashing, base64 encoding, and RC6 encryption (Dolmans and Katz, 2013; Maor, 2013; Matrosov et al., 2011).

5. Analysis of malware behaviours

The Pharos Static Binary Analysis Framework (SEI CMU, 2017) is an open source program for the static analysis of malware. Pharos is based on the Rose compiler infrastructure (Quinlan and Liao, 2011) which is used for disassembly, control flow analysis and instruction semantics. A major new version of the Pharos Framework was released in June 2017 and this research is based on the facilities provided by this new version. The ApiAnalyzer component of the Pharos Framework provides the capability to identify malware behaviours by use of user supplied rules that contain sequences of API calls and the relationships between the API calls. The behaviours which are identified by ApiAnalyzer correspond to low level actions in the MAEC model. ApiAnalyzer uses data flow analysis to verify the relationships between the API calls, i.e. that the calls are operating on the same handle.

A limitation of the current version of the Pharos Framework is that API analysis can only be performed on programs which use an import table to make API calls. The current

```
{
"Sig":{
  "Name":"ListProcesses",
  "Description":"List processes running on the system",
  "Category":"Behaviours",
  "Pattern": [
    { "API": "Kernel32.dll!CreateToolhelp32Snapshot",
      "Retn": { "Name":"HANDLE" }},
    { "API": "Kernel32.dll!Process32FirstW",
      "Args":[{ "Name":"HANDLE", "Type":"IN", "Index":0 }]},
    { "API": "Kernel32.dll!Process32NextW",
      "Args":[{ "Name":"HANDLE", "Type":"IN", "Index":0 }]}
  ]}
}
```

Fig. 1 – Rule to identify process iteration.

```
{
"Sig":{
  "Name":"InjectProcess",
  "Description":"Process injection",
  "Category":"Behaviours",
  "Pattern": [
    { "API": "Kernel32.dll!OpenProcess",
      "Retn": { "Name":"HANDLE" }},
    { "API": "Kernel32.dll!CreateRemoteThread",
      "Args":[{ "Name":"HANDLE", "Type":"IN", "Index":0 }]}
  ]}
}

}
```

Fig. 2 – Rule to identify process injection.

version of the Pharos Framework contains limited configuration data. It was necessary to create a configuration containing the stack offset and the API ordinal number for each API call used by each malware sample. The stack offset is the number of bytes passed on the stack when making a specific API call.

A limited number of example rules are provided with the Pharos Framework (SEI CMU, 2016). A rule is provided to search for `CreateToolhelp32Snapshot calls`, where the handle created by this call is passed to `Process32FirstW` and `Process32NextW` API calls. This rule is used to search for a behaviour where malware is iterating through running processes.

The Zeus, Citadel and Zeus P2P samples were unpacked using a static unpacker. The Vawtrak and Dyre samples were unpacked and the import tables rebuilt using OllyDbg and the Ollydmp plugin. The Dridex and Carberp malware families use hashed API obfuscation and are not suitable for use with the current version of ApiAnalyzer. A ReactorBot sample was dumped using Ollydmp, however this malware is able to defeat import table reconstruction. Therefore, the malware families used in this analysis are Zeus, Citadel, Vawtrak and Dyre.

In the experiments conducted for this paper, the unpacked malware samples were first examined with the IDA disassembler to ensure that API call sequences form behaviours which correspond to the rule being tested.

5.1. *Process injection rule 1*

The first rule tested was the example rule provided with the Pharos Framework (SEI CMU, 2016). This rule, shown in Fig. 1 searches for all occurrences of the `CreateToolhelp32Snapshot` API and uses dataflow analysis to locate all `Process32FirstW` and `Process32NextW` API calls which use the handle created by the `CreateToolhelp32Snapshot` call.

The results of using the process iteration rule against the selected malware samples is shown in Table 3.

Table 3 – Malware process iteration methods.

Malware	API Sequence Present	Detected
Zeus	Yes	-
Citadel	Yes	Yes
Vawtrak	Yes	-
Dyre	Yes	Yes

It is noted that the process iteration rule located the process iteration behaviour in the Citadel malware but not in the Zeus malware, despite the fact that Citadel is a Zeus variant and the process iteration functions are very similar. The Vawtrak sample contains a process iteration behaviour corresponding to the rule, but it was not detected. Discussion with the Pharos Framework developers indicates that the design philosophy of the program is to avoid false positives, however some false negatives are considered to be acceptable. The development of the program is in progress and the program design currently contains approximations which may not work for all malware families.

The rule in Fig. 2 is used to scan an unpacked malware sample and to locate instances of direction process injection. The results of using the process injection rule against the selected malware samples is shown in Table 4. The Vawtrak sample contained a process injection behaviour which is not detected. This sample uses Fastcall parameter passing (Sikorski and Honig, 2012) which is supported by the Pharos Framework, further work is required to determine the cause of this problem.

The Message Digest 5 (MD5) hashes of the samples used for experiments with ApiAnalyzer are shown in Table 5.

5.2. *Future work*

The open source Pharos Framework currently locates API calls through the static import table. It would be advantageous to

Table 4 – Malware process injection methods.

Malware	API Sequence Present	Detected
Zeus	Yes	Yes
Citadel	Yes	Yes
Vawtrak bit	Yes	-
Dyre	-	-

Table 5 – Sample hashes.

Malware	MD5 Hash
Zeus	306dd8c10a19c5998e88c7b1de520e2f
Citadel	24547d8e6028b77a5a62b3babc9264ad
Vawtrak 32bit	19f8bc63e882fbe7affccd814602638b
Dyre	cc0d5b95b8b30f99c1092b87c869c74c

add support for dynamic API loading into the Pharos Framework. Dynamic API loading is performed by `LoadLibrary` and `GetProcAddress` API calls. Support for obfuscated API loading, where the API is represented by a hash value and an in-memory DLL is parsed to obtain the API virtual address, would be valuable.

The utility of the Pharos Framework could be improved by the identification of cryptographic and compression operations. This would allow rules to be created which could express common behaviours such as "data downloaded, decrypted and decompressed, then written to the registry".

6. Conclusion

This paper provides a survey of seven banking malware families and draws together a fragmented and industry based literature to provide a coherent description of major banking malware families, their variants, relationships and source code leakages.

Banking malware were selected as the basis for this paper based on their prevalence, persitence, financial damage and on the lack of literature providing a good description of their internal operations.

A challenge for malware analysis is that while existing analysis techniques provide volumes of low level details from malware samples, there is a need to produce a high level understanding of malware behaviour. This task of bridging the semantic gap from low level detail into a high level understanding lies at the core of malware reverse engineering and is largely a manual process.

Although not all malware families use exactly the same behaviours, malware authors draw from a constrained set of techniques to build their malware. This allows malware families to be described in terms of the techniques which are used to implement these behaviours. Dynamic analysis techniques have been used to generate profiles of malware behaviour, however dynamic analysis has difficulty in achieving full code coverage.

This paper uses the example of the ApiAnalyzer program from the Pharos Framework to provide the static identification of malware behaviours. The use of static analysis techniques to automatically identify malware behaviours presents a representation of malware capability at a higher level of abstraction than has previously been available.

Acknowledgement

This research was funded in part through the Internet Commerce Security Laboratory (ICSL), a joint venture between Westpac, IBM and Federation University Australia.

REFERENCES

abuse.ch. Cridex, feodo, geodo dridex, whats next?; 2014. [Online]; Available from: https://www.abuse.ch/?p=8332. [Accessed 15 October 2017].

AhnLab. Malware analysis: Citadel. Tech. Rep.; 2012. [Online]. Available from: https://www.scribd.com/document/148064130/Citadel-Trojan-Report-eng. [Accessed 15 October 2017].

Al-Bataineh A, White G. Analysis and detection of malicious data exfiltration in web traffic. In Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on. IEEE; 2012. p. 26–31.

Alexander J. Ghost in the shell: a counter-intelligence method for spying while hiding in (or from) the kernel with apcs; 2012.

Alvarez R. Same zeus, different features; 2013. [Online]. Available from: https://blog.fortinet.com/2013/12/09/same-zeus-different-features. [Accessed 15 October 2017].

Alvarez R. Nesting doll: unwrapping vawtrak; 2015. https://www.virusbulletin.com/virusbulletin/2015/01/nesting-doll-unwrapping-vawtrak. [Accessed 15 October 2017].

Antonakakis M, Perdisci R, Nadji Y, Vasiloglou II N, Abu-Nimeh S, Lee W, et al. From throw-away traffic to bots: detecting the rise of dga-based malware. In USENIX Security Symposium; 2012. p. 491–506.

Aronov I. An example of common string and payload obfuscation techniques in malware; 2015. [Online]. Available from: https://securityintelligence.com/an-example-of-common-string-and-payload-obfuscation-techniques-in-malware/ [Accessed 15 October 2017].

Asinovsky P. Neverquest malware analysis; 2014. [Online]. Available from: https://devcentral.f5.com/articles/neverquest-malware-analysis. [Accessed 15 October 2017].

Author Unknown. Powerloader injection – something truely amazing; 2013. [Online]. Available from: http://www.malwaretech.com/2013/08/powerloader-injection-something-truly.html. [Accessed 15 October 2017].

AV-Test. Av-test security report 2016/2017. Tech. Rep.; 2017. [Online]. Available from: https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf. [Accessed 15 October 2017].

Barabosch T, Eschweiler S, Gerhards-Padilla E. Bee master: detecting host-based code injection attacks. Detection of intrusions and malware, and vulnerability assessment. Springer; 2014. p. 235–54.

Baumhof A, Shipp A. Zeus trojan update new variants based on leaked source code; 2011. [Online]. Available from: http://www.tidos-group.com/blog/2011/09/30/zeus-trojan-update-new-variants-based-on-leaked-zeus-source-code/. [Accessed 15 October 2017].

Berdajs J, Bosnić Z. Extending applications using an advanced approach to dll injection and api hooking. Softw Prac Exp 2010;40(7):567–84.

Binsalleeh H, Ormerod T, Boukhtouta A, Sinha P, Youssef A, Debbabi M, et al. On the analysis of the zeus botnet crimeware toolkit," in Privacy Security and Trust (PST). 2010 Eighth Annual International Conference on. IEEE; 2010. p. 31–8.

Bitdefender. Tracking rovnix; 2014. [Online]. Available from: http://www.malwaredigger.com/2015/06/rovnix-payload-and-plugin-analysis.html. [Accessed 15 October 2017].

Blueliv. Chasing cybercrime: network insights of dyre and dridex trojan bankers; 2015.

Blunden B. The rootkit arsenal: escape and evasion in the dark corners of the system. Jones & Bartlett Publishers; 2012.

Born K. Browser-based covert data exfiltration; 2010. arXiv preprint arXiv: 1004.4357.

Boutin J-I. The evolution of webinjects; 2014. [Online]. Available from: https://www.virusbulletin.com/conference/vb2014/abstracts/evolution-webinjects. [Accessed 15 October 2017].

Brand M. Analysis avoidance techniques of malicious software; 2010 [Ph.D. dissertation]. Edith Cowan University.

Brey BB. Intel microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, pentium, pentium proprocessor, pentium ii, iii, 4; 2005.

Cannell J. Obfuscation: malware's best friend; 2015. [Online]. Available from: https://blog.malwarebytes.org/threat-analysis/2013/03/obfuscation-malwares-best-friend/. [Accessed 15 October 2017].

Carrier B. File system forensic analysis. Addison-Wesley Reading; 2005.

Carvey H. The windows registry as a forensic resource. Digit Invest 2005;2(3):201–5.

Carvey H. Windows forensic analysis DVD toolkit. Syngress; 2009.

Certeza RA. Dealing with the mess of dridex; 2015. [Online]. Available from: http://www.trendmicro.com/vinfo/us/threat-encyclopedia/web-attack/3147/dealing-with-the-mess-of-dridex. [Accessed 15 October 2017].

Cherepanrov A, Lipovsky R. Hesperbot: a new, advanced banking trojan in the wild; 2013. [Online]. Available from: https://www.welivesecurity.com/wp-content/uploads/2013/09/Hesperbot_Whitepaper.pdf. [Accessed 15 October 2017].

Chiu A, Villegas A. Threat spotlight: dyre/dyreza: an analysis to discover the dga; 2015. [Online]. Available from: http://blogs.cisco.com/security/talos/threat-spotlight-dyre. [Accessed 15 October 2017].

Christodorescu M, Jha S, Seshia S, Song D, Bryant RE. Semantics aware malware detection. In Security and Privacy, 2005 IEEE Symposium on. IEEE; 2005. p. 32–46.

Christodorescu M, Jha S, Kruegel C. Mining specifications of malicious behavior. In Proceedings of the 1st India software engineering conference. ACM; 2008. p. 5–14.

Dietrich C. Through the window: creative code invocation; 2014. [Online]. Available from: https://www.crowdstrike.com/blog/through-window-creative-code-invocation. [Accessed 15 October 2017].

Dolmans R, Katz W. Rp1: Carberp malware analysis; 2013.

Dos Santos J. Troyan citadel backconnect vnc server manager; 2012. [Online]. Available from: http://laboratoriomalware.blogspot.de/2012/12/troyan-citadel-backconnect-windows.html. [Accessed 15 October 2017].

Falliere N, Chien E. Zeus: King of the bots; 2009.

Feng C. The evolution of rovnix: new virtual file system (vfs); 2014. [Online]. Available from: http://blogs.technet.com/b/mmpc/archive/2014/05/05/the-evolution-of-rovnix-new-virtual-file-system-vfs.aspx. [Accessed 15 October 2017].

Forrester J. An exploration into the use of webinjects by financial malware; 2014 Master's thesis]; Rhodes University.

Garcia-Cervigon M, Llinas MM. Browser function calls modeling for banking malware detection. In Risk and Security of Internet and Systems (CRiSIS), 2012 7th International Conference on. IEEE; 2012. p. 1–7.

Giuliani M, Allievi A. Carberp – a modular information stealing trojan; 2010.

Grégio ARA, Afonso VM, Fernandes Filho DS, de Geus PL, Jino M. Toward a taxonomy of malware behaviors. Comput J 2015;58(10):2758–77.

Grill B, Platzer C, Eckel J. A practical approach for generic bootkit detection and prevention. In Proceedings of the Seventh European Workshop on System Security. ACM; 2014. p. 4.

Hanel A. Dyre infection analysis by alexander hanel; 2014. [Online]. Available from: https://rstforums.com/forum/topic/89344-dyre-infection-analysis-by-alexander-hanel. [Accessed 15 October 2017].

Harley D. ESET. Hasta la vista, bootkit: exploiting the vbr; 2011. [Online]. Available from: www.welivesecurity.com/2011/08/23/hasta-la-vista-bootkit-exploiting-the-vbr. [Accessed 15 October 2017].

Hunt G, Brubacher D. Detours: binary interception of win32 functions. In Usenix Windows NT Symposium; 1999. p. 135–43.

Huss D, Matthew M. Proofpoint. In the shadows: Vawtrak aims to get stealthier by adding new data cloaking; 2015. [Online]. Available from: https://www.proofpoint.com/us/threat-insight/post/In-The-Shadows. [Accessed 15 October 2017].

Ibsen J. aplib v1.1.1 - compression library; 2014. [Online]. Available from: http://ibsensoftware.com/products_aPLib.html. [Accessed 15 October 2017].

Jackson D. SecureWorks. Gozi trojan; 2007. [Online]. Available from: http://www.secureworks.com/cyber-threat-intelligence/threats/gozi/. [Accessed 15 October 2017].

Kalnai P. Banking trojan carberp: an epitaph?; 2013. [Online]. Available from: https://blog.avast.com/2013/04/08/carberp_epitaph/. [Accessed 15 October 2017].

Kasslin K, Ståhlberg M, Larvala S, Tikkanen A. Hide'n seek revisited–full stealth is back. In Proceedings of the 15th Virus Bulletin International Conference, 2005.

Kharouni L. Automating online banking fraud. Technical Report, Trend Micro Incorporated, Tech. Rep.; 2012.

Kilman D. Decoding vawtrak neverquest traffic; 2014. [Online]. Available from: http://cybersecuritymave-techie.blogspot.com.au/2014/07/decoding-vawtrakneverquest-traffic.html. [Accessed 15 October 2017].

Kimberley. Analysis of dyreza – changes & network traffic; 2014a. [Online]. Available from: http://stopmalvertising.com/malware-reports/analysis-of-dyreza-changes-network-traffic.html. [Accessed 15 October 2017].

Kimberley. Introduction to dyreza, the banker that bypasses ssl; 2014b. [Online]. Available from: http://stopmalvertising.com/malware-reports/introduction-to-dyreza-the-banker-that-bypasses-ssl.html. [Accessed 15 October 2017].

Kimberly. Analysis of dridex / cridex / feodo / bugat; 2014a. [Online]. Available from: http://stopmalvertising.com/malware-reports/analysis-of-dridex-cridex-feodo-bugat.html. [Accessed 15 October 2017].

Kimberly. Analysis of vawtrak; 2014b. [Online]. Available from: http://stopmalvertising.com/malware-reports/analysis-of-vawtrak.html. [Accessed 15 October 2017].

Kirillov I, Beck D, Chase P. Mitre Corp. Maec default vocabularies specification, version 4.1; 2014a. [Online]. Available from: https://maec.mitre.org/language/version4.1/MAEC_Vocabs_Spec_v1_1.pdf. [Accessed 15 October 2017].

Kirillov I, Beck D, Chase P. Mitre Corp. The maec language, overview; 2014b. [Online]. Available from: http://maecproject.github.io/documentation. [Accessed 15 October 2017].

Kolbitsch C, Holz T, Kruegel C, Kirda E. Inspector gadget: automated extraction of proprietary gadgets from malware binaries. In Security and Privacy (SP), 2010 IEEE Symposium on. IEEE; 2010. p. 29–44.

Kovacs E. SecurityWeek, Tech. Rep.. Dyre banking trojan counts processor cores to detect sandboxes; 2015.

Kroustek J. AVG, Tech. Rep.. Analysis of banking trojan vawtrak; 2015. [Online]. Available from: http://now.avg.com/wp-content/uploads/2015/03/avg_technologies_vawtrak_banking_trojan_report.pdf. [Accessed 15 October 2017].

Kruse P. Carberp source code confirmed leaked; 2013. [Online]. Available from: https://www.csis.dk/en/csis/news/3961/. [Accessed 15 October 2017].

Kuhn J, Mueller L, Kessem L. The dyre wolf: attacks on corporate banking accounts; 2015. [Online]. Available from: https://portal.sec.ibm.com/mss/html/en_US/support_resources/pdf/Dyre_Wolf_MSS_Threat_Report.pdf. [Accessed 15 October 2017].

Lau B, Svajcer V. Measuring virtual machine detection in malware using dsd tracer. J Comp Virol 2010;6(3):181–95.

Lee A, Varadharajan V, Tupakula U. On malware characterization and attack classification. In Proceedings of the First Australasian Web Conference-Volume 144. Australian Computer Society, Inc.; 2013. p. 43–7.

Lemos R. Dyre malware developers add code to elude detection by analysis tools; 2015. [Online]. Available from: http://www.eweek.com/security/dyre-malware-developers-add-code-to-elude-detection-by-analysis-tools.html. [Accessed 15 October 2017].

Liang Z, Yin H, Song D. Hookfinder: identifying and understanding malware hooking behaviors. Department of Electrical and Computing Engineering; 2008. p. 41.

Ligh M. Secure Science Corporation, Tech. Rep.. [prg] malware case study; 2006.

Malware Digger. Rovnix dropper analysis (trojandropper:win32/rovnix.p); 2015a. [Online]. Available from: http://www.malwaredigger.com/2015/05/rovnix-dropper-analysis.html. [Accessed 15 October 2017].

Malware Digger. Rovnix payload analysis; 2015b. [Online]. Available from: http://www.malwaredigger.com/2015/06/rovnix-payload-and-plugin-analysis.html. [Accessed 15 October 2017].

Mankin J. Classification of malware persistence mechanisms using low-artifact disk instrumentation; 2013 Ph.D. dissertation; Northeastern University Boston.

Maor E. Carberp source code for sale – bootkit included!; 2013. [Online]. Available from: http://securityintelligence.com/carberp-source-code-sale-free-bootkit-included/#.VamduZP5s_s. [Accessed 15 October 2017].

Marcos M. Trend Micro. New dyre variant hijacks microsoft outlook, expands targeted banks; 2015. [Online]. Available from: http://blog.trendmicro.com/trendlabs-security-intelligence/new-dyre-variant-hijacks-microsoft-outlook-expands-targeted-banks. [Accessed 15 October 2017].

Martignoni L, Stinson E, Fredrikson M, Jha S, Mitchell JC. A layered architecture for detecting malicious behaviors. Recent advances in intrusion detection. Springer; 2008. p. 78–97.

Matrosov A. Rovnix bootkit framework updated; 2012. [Online]. Available from: http://www.welivesecurity.com/2012/07/13/rovnix-bootkit-framework-updated/. [Accessed 15 October 2017].

Matrosov A, Rodionov E, Volkov D, Harley D. Win32/carberp when you're in a black hole stop digging; 2011. [Online]. Available from: https://www.eset.com/ca/business/resources/white-papers/win32carberp-when-youre-in-a-black-hole-stop-digging. [Accessed 15 October 2017].

Microsoft. Tech. Rep.. Developer resources: Api index; 2016. [Online]. Available from: https://developer.microsoft.com/en-us/windows. [Accessed 15 October 2017].

Milletary J. Citadel trojan malware analysis; 2012. [Online]. Available from: http://botnetlegalnotice.com/citadel/files/Patel/Decl/Ex20.pdf. [Accessed 15 October 2017].

Mohaisen A, Alrawi O, Mohaisen M. Amal: high-fidelity, behavior-based automated malware analysis and classification. Comp Sec 2015;52:251–66.

O'Brien D. Symantec. Dridex: tidal waves of spam pushing dangerous financial trojan; 2014. [Online]. Available from: www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/dridex-financial-trojan.pdf. [Accessed 15 October 2017].

Plohmann D, Yakdan K, Klatt M, Bader J, Gerhards-Padilla E. A comprehensive measurement study of domain generating malware. In USENIX Security Symposium. USENIX; 2016. p. 263–78.

Quinlan D, Liao C. The rose source-to-source compiler infrastructure. In Cetus users and compiler infrastructure workshop, in conjunction with PACT, vol. 2011; 2011. p. 1.

Quist D, Smith V. Offensive Computing. Detecting the presence of virtual machines using the local data table; 2006.

Rahimian A, Ziarati R, Preda S, Debbabi M. On the reverse engineering of the citadel botnet. Foundations and practice of security. Springer; 2014. p. 408–25.

Riccardi M, Di Pietro R, Vila JA. Taming zeus by leveraging its own crypto internals. In eCrime Researchers Summit (eCrime), 2011. IEEE; 2011. p. 1–9.

Rocha L. Malware analysis – dridex loader – part 2; 2016. [Online]. Available from: https://countuponsecurity.com/2016/08/28/malware-analysis-dridex-loader-part-2. [Accessed 15 October 2017].

Rodionov DE, Matrosov A, Harley D. In VB Conference. Bootkits: past, present & future; 2014. [Online]. Available from: http://static5.esetstatic.com/us/resources/white-papers/RodionovMatrosov-VB2012.pdf. [Accessed 15 October 2017].

Rodionov E, Matrosov A. ESET. Defeating anti-forensics in contemporary complex threats; 2012. [Online]. Available from: http://go.eset.com/us/resources/white-papers/RodionovMatrosov-VB2012.pdf. [Accessed 15 October 2017].

Rossow C, Dietrich C, Bos H. Large-scale analysis of malware downloaders. Detection of intrusions and malware, and vulnerability assessment. Springer; 2012. p. 42–61.

Sanghavi M. Symantec. Dridex and how to overcome it; 2016. [Online]. Available from: https://www.symantec.com/connect/blogs/dridex-and-how-overcome-it. [Accessed 15 October 2017].

Scanlon LJ. Assembly language programming for the IBM PC AT. Brady Communications Company; 1986.

Selvaraj K. A brief look at zeus/zbot 2.0; 2010. [Online]. Available from: http://www.symantec.com/connect/blogs/brief-look-zeuszbot-20. [Accessed 15 October 2017].

SEI CMU. Static identification of program behavior using sequences of api calls; 2016. [Online]. Available from: https://insights.sei.cmu.edu/sei_blog/2016/04/static-identification-of-program-behavior-using-sequences-of-api-calls.html. [Accessed 15 October 2017].

SEI CMU. Pharos static binary analysis framework; 2017. [Online]. Available from: https://github.com/cmu-sei/pharos. [Accessed 15 October 2017].

Shulman A, Dorfman H. Dyre financial malware internals; 2015. [Online]. Available from: https://devcentral.f5.com/d/dyre-malware-internals. [Accessed 15 October 2017].

Sikorski M, Honig A. Practical malware analysis: the hands-on guide to dissecting malicious software. No Starch Press; 2012.

Singh PK. A physiological decomposition of virus and worm programs; 2002 Master's thesis; University of Louisiana at Lafayette.

Sood A, Enbody R. Targeted cyber attacks: multi-staged attacks driven by exploits and malware. Syngress; 2014.

Sood A, Rohit B. Virus Bulletin. Prosecting the citadel botnet – revealing the dominance of the zeus descendent: part one; 2014. [Online]. Available from: https://www.virusbulletin.com/virusbulletin/2014/09/prosecting-citadel-botnet-revealing-dominance-zeus-descendent-part-one. [Accessed 15 October 2017].

Stevens K, Jackson D. Zeus banking trojan report; 2010. [Online]. Available from: http://www.secureworks.com/cyber-threat-intelligence/threats/zeus/. [Accessed 15 October 2017].

Stone-Gross B, Khandhar P. Dyre banking trojan; 2014. [Online]. Available from: http://www.secureworks.com/cyber-threat-intelligence/threats/dyre-banking-trojan/. [Accessed 15 October 2017].

Su M. Virus Bulletin. Dridex in the wild; 2015. [Online]. Available from: https://www.virusbulletin.com/virusbulletin/2015/07/dridex-wild. [Accessed 15 October 2017].

Suenaga M. A museum of api obfuscation on win32. In Proceedings of 12th Association of Anti-Virus Asia Researchers International Conference, AVAR, 2009.

Sun H-M, Tseng Y-T, Lin Y-H, Chiang T. In 2006 International Computer Symposium, ICS. Detecting the code injection by hooking system calls in windows kernel mode; 2006.

Symantec Security Response. Dyre: emerging threat on financial fraud landscape; 2015. [Online]. Available from: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/dyre-emerging-threat.pdf. [Accessed 15 October 2017].

Thadani R. Report: the dridex trojan is back; 2016. [Online]. Available from: http://blogs.quickheal.com/report-the-dridex-trojan-is-back. [Accessed 15 October 2017].

Tigzy. Carberp bootkit : how self protection is effective; 2013. [Online]. Available from: http://www.adlice.com/carberp-bootkit-how-self-protection-is-effective/. [Accessed 15 October 2017].

Trend Micro. Trend Micro, Tech. Rep.. File-patching zbot variants zeus 2.0 levels up; 2010.

Trend Micro. A closer look at dyre malware, part 1; 2014. [Online]. Available from: http://blog.trendmicro.com/trendlabs-security-intelligence/a-closer-look-at-dyre-malware-part-1/. [Accessed 15 October 2017].

Trend Micro. Banking malware vawtrak now uses malicious macros, abuses windows powershell; 2015.

Trusteer. Carberp under the hood of carberp: malware & configuration analysis; 2010. [Online]. Available from: http://www.trusteer.com/sites/default/files/Carberp_Analysis.pdf. [Accessed 15 October 2017].

Tsaur W, Chen Y-C. Exploring rootkit detectors' vulnerabilities using a new windows hidden driver based rootkit. In Social Computing (SocialCom), 2010 IEEE Second International Conference on. IEEE; 2010. p. 842–8.

US Air Force. Analysis of the intel pentiums ability to support a secure virtual machine monitor. In Proceedings of the… USENIX Security Symposium. USENIX Association; 2000. p. 129.

Wang Y-M, Roussev R, Verbowski C, Johnson A, Wu M-W, Huang Y, et al. Gatekeeper: monitoring auto-start extensibility points (aseps) for spyware management. in LISA, 4; 2000. p. 33–46.

Wang Y-M, Beck D, Vo B, Roussev R, Verbowski C. Detecting stealth software with strider ghostbuster. In Dependable Systems and Networks, 2005. DSN 2005. In Proceedings. International Conference on. IEEE; 2005. p. 368–77.

Wyke J. Sophos Labs, Tech. Rep.. What is zeus?; 2011.

Wyke J. Sophos Labs Naked Security Blog, Tech. Rep.. The citadel crimeware kit – under the microsope; 2012.

Wyke J. Vawtrak – international crimeware-as-a-service; 2014.

Wyke J. Technical Report, Sophos, Tech. Rep.. Breaking the bank(er): Automated configuration data extraction for banking malware; 2015.

Zeus Author. Zeus source code; 2011. [Online]. Available from: https://github.com/Visgean/Zeus. [Accessed 15 October 2017].

Paul Black is studying a PhD in Information Security at the Internet Commerce Security Lab (ICSL) at Federation University. His PhD topic is Techniques for the Reverse Engineering of Banking Malware. Paul has a Masters of Computing, his research topic was the reversing of Zeus malware. Paul started his career as a programmer in 1981 and has worked in banking, defence, law enforcement and malware analysis.

Iqbal Gondal is a leading researcher in the area of condition monitoring, sensor information processing, wireless communication and cyber security. Currently he is Director of Internet Commerce Security Lab (ICSL), Federation University Australia. ICSL conducts research in the application of advance analytics techniques for cybersecurity and condition monitoring and provides innovative Cybersecurity solutions to the industry. In the past, he was director of ICT strategy for the faculty of IT in Monash. He has served in the capacity of Director of Postgraduate studies for six years, member faculty board and member of Monash academic board. He is Fellow of Engineers Australia.

Robert Layton is a Data Scientist working with text problems in a number of domains. His research focuses on the methods used to build cybercrime attacks and the analysis of the outcomes. He is an Honorary Research Fellow at Federation University Australia and the inaugural Federation University Young Alumni of the Year in 2014.

Chapter 4

Reanimating Historic Malware Samples

Banking infrastructure is constructed using various operating systems and is configured to resist attack. The computer systems of banking customers have, at best, a limited security configuration. As a result, banking customer computer systems provide a convenient vector for attack, and this chapter provides research supporting the dynamic analysis of the full capabilities of these malware families.

Dynamic analysis is commonly used for the extraction of features from historic malware samples. Malware operations are disrupted by requesting the removal of the DNS servers of the malware C2 domains. Historic malware samples execute in a VM until access to the malware C2 network servers is requested. These malware samples respond to the absence of C2 servers by retrying the connection or by exiting. As a result, the command interface of historic malware samples is no longer controlled. This results in the extraction of features that differ from those that would be extracted in-the-wild, thus invalidating the results of machine learning based research using these features. A solution to this problem is to create C2 server emulators for the required malware families. The process of building a C2 emulator identifies the command protocol of the malware sample. The construction of the C2 emulator provides a test environment that allows the full capabilities of the malware sample to be exercised and accurate behavioural profiles to be created. C2 server construction techniques are limited to manual or semi-automated methods because fully automated construction of C2 server emulators is not possible, and this technique does not scale to all malware samples.

This chapter provides examples of techniques for the creation of C2 server emulators for three malware families (Zeus, CryptoWall and CryptoLocker). Zeus banking malware was selected as it is representative of banking malware families, and the leaked source code assisted C2 emulator construction. The Cryptowall and CryptoLocker C2 emulators were selected to support ransomware identification research. This chapter illustrates C2 emulator construction using manual reverse engineering techniques, and provides a review of semi-automated techniques for the construction of C2 server emulators.

The work in this chapter has been accepted as a book chapter for publication:

- P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, ”Reanimating Historic Malware Samples”, Book Chapter in *Malware Analysis using Artificial Intelligence and Deep Learning.* Springer, 2020.

4.1 Introduction

Many types of malware, including information-stealing malware, ransomware, and Remote Access Trojans (RATs) are controlled from an attacker's Command and Control (C2) servers [63]. Anti-virus organisations seek to defeat malware attacks by requesting removal of C2 server Domain Name Server (DNS) records. For discussion, a historical malware sample is defined as a malware sample that has had its C2 servers removed. Large datasets of historical malware samples are available for academic experiments. However, due to the age of these malware samples, their C2 servers are no longer available. To cope with high volumes of malware production, malware analysis is increasingly performed using machine learning techniques [64]. Dynamic analysis is commonly used for feature extraction. However, due to the absence of their C2 servers, after initialization, malware execution may exit, or loop attempting to establish C2 server connections. As a result, the command interface of historic malware samples is no longer controlled. This results in the extraction of features that differ from those that would be extracted in-the-wild, thus invalidating the results of machine learning research based on these features.

It is noted that research techniques exist for automatic protocol analysis of malware [35]. However, these techniques depend on malware communications with live C2 servers. The usage of the malware capabilities is determined by the malware operators, and live testing may not reveal the full extent of the malware's capability. Other issues related to performing research with live malware include difficulties in obtaining a consistent supply of live malware, unknown configuration, unknown triggering conditions, detection of the analysis IP address (mitigated by the use of an anonymizing proxy), or the malware operators gaining access to the analysis VM via malware provided interfaces.

Researchers have recognised the need to prevent malware experiments from causing harm on the internet. Research systems have been built to provide containment of malware research [36]. However, these systems do not address the C2 server problems faced when performing experiments with historical datasets. Internet simulator programs [37] may be used as part of a malware analysis environment and can provide generic responses to requests for common internet services. A malware process may request a connection to a common website to perform a connectivity check, and an internet simulator may be able to satisfy this request. However, if a connection to a C2 server or other attacker-controlled infrastructure is requested, an internet simulator will not be able to respond with the protocol required by the malware.

The Botnet Evaluation Environment (BEE) provides an isolated environment for botnet research with emulated C2 servers for execution of the Agobot, SDBot, GTBot, Phatbot, and Spybot malware [38]. An isolated Waledac botnet was created by reverse engineering the Waledac malware and identifying the Waledac botnet protocol. An emulated C2 server was built to support this protocol, and a 3000 node Waledac botnet was built. This isolated botnet was used to research security vulnerabilities that could be used to take down the Waledac botnet [39].

To illustrate the benefits of building C2 server emulators for machine learning purposes, this chapter provides examples of techniques for the creation of C2 server emulators for three malware families (Zeus, CryptoWall and CryptoLocker) using manual reverse engineering techniques. [1] This chapter also provides a review of semi-automated techniques for the construction of C2

[1]The datasets and code related to this research are available on request from the corresponding author.

server emulators.

4.1.1 Motivation

At a high level, the need to build a C2 server emulator will be the result of the following requirements:

- The need to perform research using historical malware samples,
- The need to control the full capabilities of a malware sample,
- The need to perform the research in an isolated environment.

The construction of C2 server emulators has the following benefits: the ability to control the malware through its network interface allows the execution of the full capabilities of the malware and the extraction of features that would otherwise not be possible using historical malware samples. Using an emulated C2 server allows the testing of malware samples in isolation from the internet, which prevents criminal groups from becoming aware of the research.

4.1.2 Emulator Architecture

The architecture of a C2 server emulator will be similar irrespective of whether a manual or semi-automated process is used to construct the emulator. A representative C2 server emulator consists of an isolated network using two or more virtual machines (VMs). One VM (VM1) is configured to run the C2 server emulator script and a DNS simulator, while another VM (VM2) is configured to run the selected malware samples and any related programs that will be attacked by this malware. The DNS simulator resolves requests from the malware VM, and malware protocol requests are read by the C2 emulator. An illustration of this architecture is shown in Figure 4.1. C2 server emulators may be created using either manual or semi-automated construction techniques. Sections 2-4 discuss manual construction, and section 5 provides a review of semi-automated construction techniques.

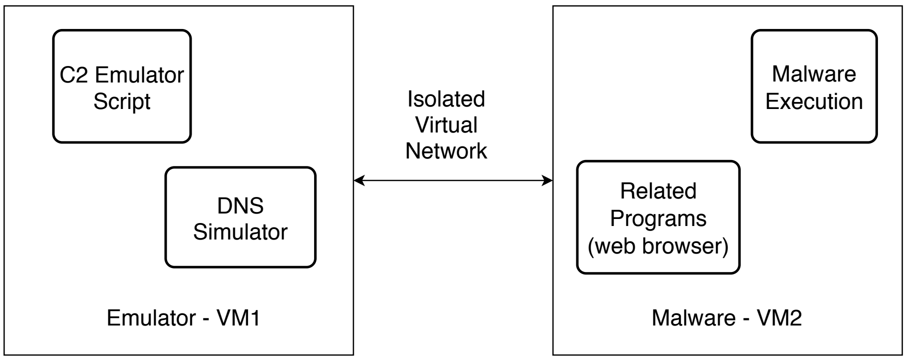


Figure 4.1: C2 Emulator Architecture

4.2 Manual Construction

The creation of C2 emulators provides a test harness that allows the full capability of historic malware samples to be controlled in an isolated network. The process for the manual construction of C2 emulators can be described in an abstract manner as a process of the guided discovery of the communication and command processing paths of a malware sample using a debugger and the corresponding iterative development of a script to generate network traffic to control this execution path. A difficulty with this high-level description of the C2 server emulator construction process is that there may be difficulties in fully understanding how to implement this process. A malware analysis environment using a manually constructed C2 emulator is described in [65]. To illustrate the manual construction process, sections 3 and 4 provide examples of the manual construction of C2 server emulators for a common information-stealing malware and two ransomware families.

The requirement for the emulation of a malware C2 server arose from a research project using machine learning for the detection of webinjects. Webinjects are malicious HTML that are injected into web browser sessions and are used to steal banking credentials and to illegally transfer funds [66]. Information stealing malware targeting banking infrastructure (banking malware, banking trojans) contain facilities for intercepting credentials prior to encryption and injecting content into internet banking sessions. This is performed by injecting malware into the browser process and gaining control of networking Application Programming Interface (API) functions with the use of user-mode hooking techniques [4]. Three options were considered for webinject generation:

- The use of live malware to perform webinjects, Zarathustra and Prometheus performed webinject detection using live malware samples [67, 68],
- The use of Java-Script methods to inject code into the browser session,
- The creation an emulated C2 server that can be used in conjunction with a historical malware sample to perform user specified webinjects.

The problems associated with using live malware for research purposes have been discussed previously. While Java-Script methods can be used to inject HTML code into a browser session, this injected HTML may differ from webinjects created by malware. The time required to perform the reverse engineering work is a significant consideration when deciding whether to build an emulated C2 server. However, this may be offset by the significant benefits of being able to control the full capabilities of the malware on an isolated network and the collection of more representative features.

The following sections provide the details of the construction of the C2 server emulators for the Zeus V2, CryptoLocker and CryptoWall malware.

4.3 Zeus C2 Server Emulator

The Zeus v2 malware was selected for this research due to familiarity with this malware from previous research. The Zeus C2 server emulator provides the capability to create custom webinjects on an isolated network and to capture the modified webpages for use in a webinject detection machine learning system [69]. In the following description, class and function names (e.g. Core::GetBaseConfig) are taken from the leaked Zeus source code [70]. A Zeus v2.1.0.1 malware sample with an MD5 hash of a2a21d66f72ee53cfbc2dcfe929ffaba was used in this research. This malware was unpacked using a custom static unpacker. The unpacked Zeus sample was loaded into the Interactive Disassembler (IDA). This IDA database was used to record the malware execution and to determine suitable API calls for breakpoints.

The Zeus v2 malware has an anti-analysis mechanism known as hardware locking. When Zeus malware infects a computer, a copy of the malware is installed in the filesystem, and a block of encrypted binary data is embedded into this installed malware. This encrypted data includes the malware's installation directory and a Globally Unique Identifier (GUID) that was generated from the computer's hard disk [71]. When a previously installed Zeus sample is executed in an analysis environment, execution on a new computer will be detected, and the malware will exit.

The hardware locking test was disabled by editing the machine code in the Core::EntryPoint function, the jump instruction controlling the call to the ExitProcess API was overwritten with no operation (NOP) instructions. This was the only change that was made to the Zeus malware.

The guiding principle in building the Zeus C2 server emulator was to perform the minimum amount of reverse engineering needed to produce a C2 server emulator. Two VMs were used where VM1 was running the python C2 server emulator and the internet simulator, and VM2 was running the Zeus malware sample and Internet Explorer version 8 for the webinjects testing. When the Zeus sample is executed, a copy of this malware is injected into the Explorer process, and the injected malware attempts to connect to the C2 server.

The Zeus configuration is an encrypted binary data structure containing text specifying the target URL, injection start pattern, and the corresponding webinjects. The Zeus configuration is created by the Zeus configuration builder, subject to the malware author's security controls. To simplify the researcher workflow, the emulated C2 server uses a simplified text based configuration containing the targeted URLs, injection start patterns, and the corresponding webinjects. An example of the simplified configuration used by the emulated Zeus C2 server is given in Figure 4.2. The C2 server emulator injects additional JavaScript to dump the DOM of the injected webpage into a shared host directory for analysis.

An initial C2 server emulator returning a response of 256 null bytes was created using the python BaseHttpServer class. On VM2, a debugger was attached to the explorer.exe process, a breakpoint was set at the InternetReadFile API, and the Zeus malware sample was executed. The emulated C2 server returned a response, and the breakpoint on the InternetReadFile API was hit in the Wininet::DownloadData function. Single stepping in the debugger was continued until the BinStorage::Unpack function was called where the

```
data_before
<title>Savings</title>
end\_before

data_inject
<p><font color="red"><b>ICSL Web Inject test framework</b></font></p>
end_inject

url_target
http://redacted.com.au/check.aspx?p=52
end_target
```

Figure 4.2: Simplified Zeus Configuration

following operations were observed:

- RC4 decryption of the received configuration,
- Recursive XOR decoding of the decrypted configuration Crypt::_visualEncrypt,
- Checking of the MD5 signature stored in the header of the decoded configuration,
- The writing of the encrypted Zeus configuration to the Windows registry.

Based on these observations, the C2 server emulator was updated to use RC4 encryption, recursive XOR encoding, and MD5 signing of the configuration data. A flowchart showing the steps involved in the creation of the encrypted Zeus configuration is given in Figure 4.3.

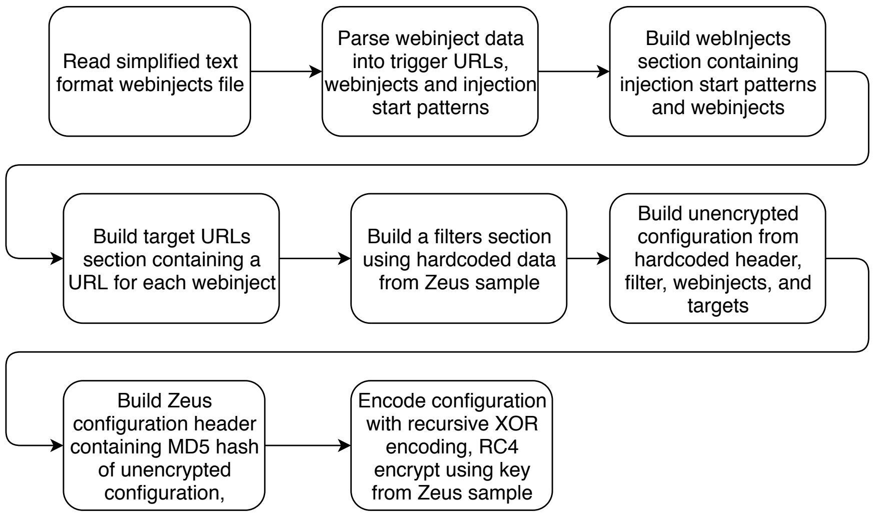

Read simplified text format webinjects file
Parse webinject data into trigger URLs, webinjects and injection start patterns
Build webInjects section containing injection start patterns and webinjects
Build target URLs section containing a URL for each webinject
Build a filters section using hardcoded data from Zeus sample
Build unencrypted configuration from hardcoded header, filter, webinjects, and targets
Build Zeus configuration header containing MD5 hash of unencrypted configuration,
Encode configuration with recursive XOR encoding, RC4 encrypt using key from Zeus sample

Figure 4.3: Creation of Encrypted Zeus Configuration

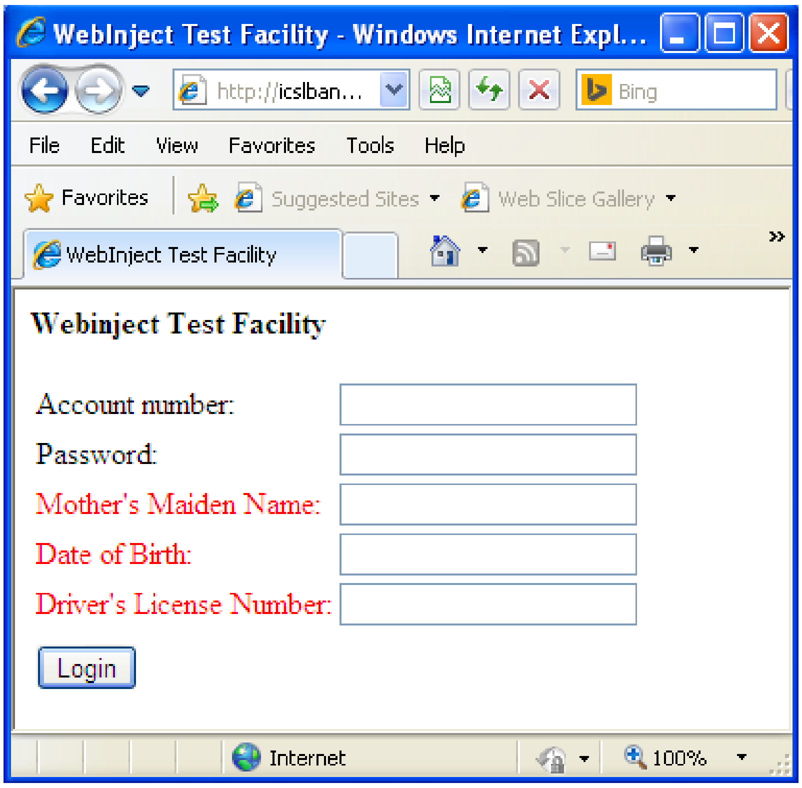
A full explanation of the Zeus configuration and webinjects processing would require excessive detail. The following provides a high-level view of the structure of the Zeus configuration and the operation of the malware in the browser. The Zeus configuration file consists of the following sections: header, filters, a number of webinject sections, and an injects list containing the targeted URLs.

Using a debugger to follow the execution of the injected Zeus code in the web browser was necessary in order to debug the processing of the encrypted Zeus configuration created by the C2 server emulator and to determine the minimum configuration sections required to allow successful webinjects. To gain control of the Zeus code in Internet Explorer, a debugger was attached to the Internet Explorer 8 parent process, and a breakpoint was set on the GetModuleHandleW API. Following the validation of the Zeus configuration in the Explorer process, Zeus malware is injected into the browser process to monitor the current webpages, to detect triggering URLs, and to perform injection of the webinjects.

When executing in the context of a web browser, Zeus hooks the browser's HttpSendRequest and InternetReadFile APIs. When the HttpSendRequest API is called, this results in a call to WininetHook::OnHttpSendRequest to check the Zeus configuration filter actions. If the filter action is not "ignore", the HTTP request is added to an HTTP connections tracking table. When the InternetReadFile API is called, this results in a call to WininetHook ::OnInternetReadFile to check the HTTP response. If this connection is in the tracked HTTP connections table, then the HttpGrabber::ExecuteInjects function is called to determine whether the URL is in the targeted URLs section, and if required injects the webinject into the HTTP response data. An example of a Zeus webinject, injected text in red, is shown in Figure 4.4.

4.4 Ransomware C2 Server Emulators

The following section provides details of the construction of a C2 server emulator for the CryptoLocker and CryptoWall ransomware. The reverse engineering of the CryptoLocker malware was straightforward, and the construction of the C2 server emulator was simple. However, the reverse engineering of the Cryp-



Webinject Test Facility

Account number:	
Password:	
Mother's Maiden Name:	
Date of Birth:	
Driver's License Number:	

Figure 4.4: Example Zeus Webinject

toWall ransomware was complicated by injection into multiple processes and API obfuscation.

4.4.1 CryptoLocker C2 Server Emulator

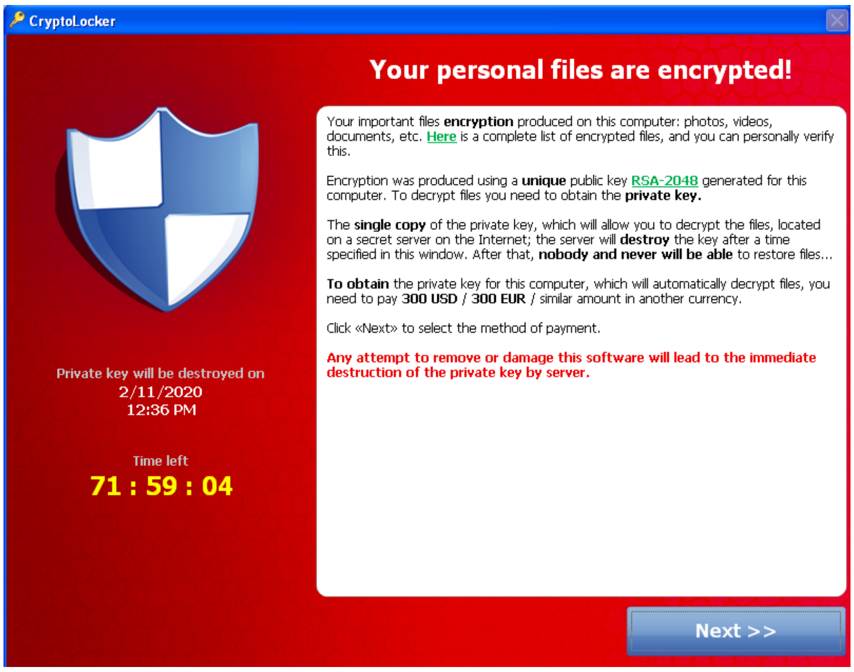
CryptoLocker ransomware was identified in 2013, and the number of infected computers is not known. The MD5 hash of the CryptoLocker sample used in this research is fec5a0d4dea87955c124f2eaa1f759f5 [72]. This sample was obtained from Malpedia [73] and includes an unpacked version of the malware. CryptoLocker uses the Microsoft CryptoAPI, which simplifies the identification of cryptographic operations. CryptoLocker encryption of communications and files uses a randomly generated AES key. This AES key is then RSA encrypted and is embedded into each encrypted object. CryptoLocker communications encryption makes use of a public key embedded in the malware and a private key stored in the C2 server. CryptoLocker file encryption uses a public key provided by the C2 server. The private key needed to decrypt the files is only provided after the ransom is paid [74, 75].

Running the unpacked malware in a debugger showed that a second malware process was started, and the first process terminated. Examination of the malware in IDA showed that the function controlling C2 server communications and user file encryption was located at address `0x40B2A1`. A shortcut to gaining control of this malware was performed by editing the first two bytes of this function in the unpacked malware to `0xEBFE`. This is a two byte loop that will cause any process executing this function to loop and will stop the malware from progressing. The looping process was identified by its high CPU usage using the task manager. The debugger was then attached to gain control of the malware. Stepping through the malware with the debugger showed that the following (before encryption) data was sent to the C2 server `"version=1& id=1&name=USERNAME-06752E85&group=sell03-10&lid=en-US"`.

The response from the C2 server is intended to be encrypted with a private key contained in the C2 server. However, the C2 servers are no longer active, and the private key is not available. Two approaches to address the missing private key are to edit the CryptoLocker malware and replace the hardcoded RSA public key with a generated public key, and use the corresponding private key in the C2 emulator, or create unencrypted responses in the C2 emulator and modify the CryptoLocker malware to no longer check the decryption status by replacing a conditional jump with NOP instructions. The latter option was selected as it was easier to implement.

The unencrypted C2 server response was read using the InternetReadFile API and was decrypted using the CryptDecrypt API. The conditional jump instruction testing the return code from the CryptDecrypt API was overwritten with a NOP instruction, allowing unencrypted C2 server responses to be processed by the CryptoLocker malware. The malware was observed to test that the last byte of the decrypted response is zero, and the C2 server emulator was updated to send a null terminated unencrypted response.

Further use of the debugger showed that a value of 1 in the first byte of the response results in a call to a function that calls the CryptDecodeObjectEx API to decode a Privacy Enhanced Mail (PEM) format public key. This public key is located at byte 3 of the response. The completed C2 server emulator reads the initial message from the malware and returns a response of 0x01, 0x00, 0x00, followed by a null terminated PEM format public key. A screenshot of the CryptoLocker ransom demand screen displayed after the user files were encrypted is shown in Figure 4.5.


CryptoLocker

Your personal files are encrypted!

Your important files **encryption** produced on this computer: photos, videos, documents, etc. Here is a complete list of encrypted files, and you can personally verify this.

Encryption was produced using a **unique** public key RSA-2048 generated for this computer. To decrypt files you need to obtain the **private key.**

The **single copy** of the private key, which will allow you to decrypt the files, located on a secret server on the Internet; the server will **destroy** the key after a time specified in this window. After that, **nobody and never will be able** to restore files...

To obtain the private key for this computer, which will automatically decrypt files, you need to pay **300 USD** / **300 EUR** / similar amount in another currency.

Click «Next» to select the method of payment.

Any attempt to remove or damage this software will lead to the immediate destruction of the private key by server.

Private key will be destroyed on
2/11/2020
12:36 PM

Time left
71 : 59 : 04

Figure 4.5: CryptoLocker Ransom Demand

4.4.2 CryptoWall C2 Server Emulator

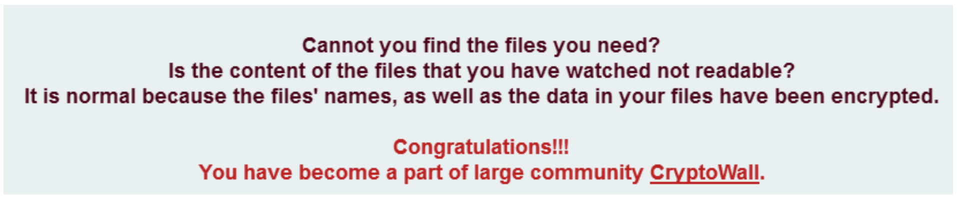
CryptoWall ransomware was identified in 2014. The MD5 hash of the CryptoWall version 4 sample used in this research is d9993ab7397f5d2a34f786b54fc55b2c. This sample was obtained from Malpedia [73] and included an unpacked version of the malware. Descriptions of the CryptoWall protocol were provided by industry blogs [72, 76, 77], this information significantly reduced the amount of reverse engineering required to build the CryptoWall C2 emulator.

Early versions of CryptoWall copied CryptoLocker's appearance, and the malware authors adopted the name CryptoWall in May 2014. CryptoWall was primarily distributed through malicious spam attachments. CryptoWall deletes volume shadow copies, and the Windows System Restore feature is disabled. CryptoWall version 4 uses a locally generated AES key to encrypt user files and filenames, RSA encryption is used to protect the AES key. CryptoWall communications are RC4 encrypted, and the RC4 cipher is passed to the C2 server in the URL of the infection announcement message [77, 76].

CryptoWall version 4 malware injects itself into a newly started `explorer` process [72]. The injected malware creates a new `svchost` process, which is injected with a copy of the malware [77]. The new `svchost` process performs ransomware operations. To gain control of the ransomware, run the analysis VM before the C2 server emulator is started. This will prevent the C2 server connection from being established and will keep the ransomware in its initialization state. Before running the CryptoWall ransomware, record a list of the process identifiers of the svchost processes. Start the CryptoWall malware and identify the new svchost process, connect to this process and set a breakpoint at the InternetConnectA API, next start the CryptoWall C2 emulator and allow the debugger to run the CryptoWall malware.

When the ransomware is executed, an HTTP POST is sent to the C2 server. The C2 server uses the sorted URL parameter as ciphertext to create an RC4 key [78]. The data passed by the HTTP POST is ASCII encoded binary data, which is decoded using the Python binascii.unhexlify function. The decoded data is decrypted using the RC4 key. The decrypted request is `"{1|crypt13001|32DC0066DCE410C9285635F121811FB99|1|2|1}"`.

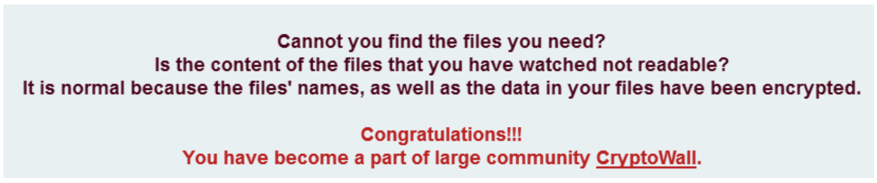
The C2 server responds by sending an RC4 encrypted response e.g. `"{204|1}"` to the infected computer. The CryptoWall ransomware responds by sending a public key request to the C2 server. The C2 server responds with a message containing a public key and a base64 encoded ransom demand graphic. When this C2 response is received, the ransomware process scans the infected computer's storage and encrypts user files. When the user files have been encrypted, an infection notification message e.g. `"{260|1}"`, is sent to the C2 server. Finally, a window is displayed on the infected computer to demand payment [76, 72]. The messages exchanged between the ransomware and the C2 server emulation are shown in Figure 4.6.


Cannot you find the files you need?
Is the content of the files that you have watched not readable?
It is normal because the files' names, as well as the data in your files have been encrypted.

Congratulations!!!
You have become a part of large community CryptoWall.

Figure 4.6: CryptoWall Messages

The CryptoWall C2 emulator implements the CryptoWall protocol that allows the ransomware to exercise its full capabilities. A screenshot of a section of the CryptoWall ransom demand screen displayed after the user files were encrypted is shown in Figure 4.7.



Cannot you find the files you need?
Is the content of the files that you have watched not readable?
It is normal because the files' names, as well as the data in your files have been encrypted.

Congratulations!!!
You have become a part of large community CryptoWall.

Figure 4.7: CryptoWall Ransom Demand

4.5 Semi-Automated Generation of C2 Server Emulators

While the ability to automatically generate C2 server emulators for arbitrary malware families would be useful, this is not currently feasible, and the recent work in literature is a semi-manual construction process.

The Imaginary C2 program [40] converts captured network traffic into request definitions that allow C2 HTTP response to be replayed. However, this traffic replay approach is not suitable for situations where initial network traffic samples are not available.

One automation approach for the creation of C2 server emulators is provided in [41]. This research refers to C2 server emulators as Custom Impersonators. Malware samples are executed on a QEMU VM, and instruction traces are collected using DECAF [42], and the instruction traces are translated into VINE intermediate language [27]. Symbolic execution [43] is performed on the instruction traces, and symbolic variables are assigned to network input. A Simple Theorem Prover (STP) constraint solver [44] is used to determine the values that determine the outcome of the control flow tests. These values can be used to identify malware control dependencies controlled by values in the network input [41]. The malware control flow graph and control dependencies are provided to assist analysts with the manual construction of C2 server emulators.

Research using ANGR [45], an open source symbolic execution framework, creates a technique that employs static analysis to determine the C2 command protocol and associated commands implemented in a common RAT. The top-level command processing function of the RAT is analysed, and for each explored path, a list of the malware API calls and their arguments, function call relationships, and the network data required to trigger the path's execution are provided [46]. Windows API models and support for the `stdcall` calling convention were added to ANGR in order to support the analysis of Windows malware. Heuristics were created to limit the number of paths explored by the symbolic execution in order to prevent potential path explosion problems. Symbolic execution commences at the manually selected Symbolic Execution Point (SEP), an execution context is needed to provide precondition values that are generated in malware initialization. In this research, the execution context was generated using two different techniques: by performing concrete execution, setting a breakpoint at the SEP, taking a memory dump and extracting the necessary parts of the execution context, or by moving the SEP backward, allowing initialization of execution context values. Symbolic execution was used to explore the command processing loop. The report produced by this technique showed the API calls, and the functions called in processing each command, as well as the network data required to trigger the processing of each command. This research targets analysis for a single RAT, requires manual SEP identification, does not support analysis of encrypted protocols and does not support mining of the analysis report from the tool output [46].

The S2E symbolic execution engine is used as the basis of research that constructs C2 servers for RATs [6]. The S2E engine performs symbolic execution of instructions and forks execution when branches are taken. An SMT solver is used to evaluate expressions and obtain concrete values. To prevent performance problems and scalability issues due to path explosion, an analyst must provide the location of the command processing loop and details of how to reach this address. The process used in this research can be summarised as Trace Generation, Trace Analysis, Speculation, Validation, and C2 Server Generation. Trace generation uses symbolic execution to explore execution paths and to maximize code coverage. A number of the recorded traces will cover the RAT command processing code. The branches taken and API execution details are recorded. Trace analysis builds Augmented Prefix Tree Acceptors (APTA) that captures API execution and branches taken along the explored paths. APTA's are Deterministic Finite Automata (DFA) that have been used in the protocol reverse engineering [47]. The goal of speculation is to generate a small number of paths that cover all of the commands. Speculative edges are added to the APTAs in an attempt to combine symbolically executed command fragments into paths containing multiple commands. The symbolic execution engine is then used to validate speculatively generated paths, when speculative edges are validated; the branches and API calls are recorded. C2 server generation is performed for each validated path that contains multiple commands. This research generates a C2 server from the code of a small RAT created for research purposes [6]. Due to the requirement for manual analysis to provide the location of the command processing loop as a starting point, this research is classified as semi-automated.

4.6 Limitations

Irrespective of the C2 emulator construction technique, some malware samples require minor modification before they can be executed with a C2 server emulator. Examples of the modifications required to allow the Zeus V2 and CryptoLocker malware to run with C2 server emulators are given below.

Zeus v2 is a self-modifying malware with a hardware locking feature that only allows the installed Zeus malware to execute on the computer that it was installed on. In the Zeus C2 server emulator, the Zeus malware was unpacked using a static unpacker, and the jump instruction that controls the hardware locking test was overwritten with NOP instructions to prevent the malware from terminating.

The CryptoLocker malware contains a hard-coded RSA public key, and the C2 server emulator is expected to respond with communications encrypted with the corresponding private key. Due to the removal of the original C2 servers, this private key is no longer available. The C2 server emulator was developed to return unencrypted responses, and the unpacked CryptoLocker malware sample was modified to skip the successful decryption check. The modified CryptoLocker sample operates in the same manner as the original ransomware, it connected to the emulated C2 server, scanned the hard disk for user files, performed file encryption and displayed the ransom demand window.

In both of these cases, a modification of the malware's machine code allowed a historical malware sample to operate at a high level of fidelity with an emulated C2 server, allowing the collection of feature sets that are comparable with malware execution in-the-wild. It is acknowledged that the technique of manually building an emulated C2 server cannot currently be performed at scale. However, cases exist where manually building a C2 server emulator allows academic research projects to be performed that would not otherwise be possible.

A limitation in the manual construction of C2 emulators is the need for a skilled analyst to perform manual reverse engineering. Techniques for the semi-automated generation of C2 server emulators do exist. However, the fully automatic generation of C2 emulators is not currently feasible due to current limitations in symbolic execution techniques.

4.7 Conclusion

Academic malware datasets consist of collections of historic malware samples. The C2 servers of these malware samples no longer exist, and when executed on a VM, these malware samples perform their initialization functions and then wait for C2 server connections that no longer exist. This initialization-only behaviour of historic malware samples provides more limited features than would be collected when the malware was running in-the-wild. Historic malware samples running without an emulated environment cannot perform many of the malware's original capabilities.

Live malware samples with active C2 servers have been used for research [67]. This approach is feasible but uncertain due to problems associated with the short lifespan of malware C2 servers, unknown malware configuration, the malware being controlled by the malware operator, and the possibility of the malware operator becoming aware of the research.

The creation of C2 server emulators allows the full capabilities of malware samples to be fully controlled by researchers in an isolated network. In the case of historic malware samples, the use of C2 server emulators allows control of malware capabilities that would no longer be available without emulation. This chapter discussed methods for the manual reverse engineering of the malware sample's command protocol and created an emulated C2 server that can control the full command interface of the malware. Three examples of methods used for the construction of emulated command and control servers were provided. Apart from the generation of C2 server emulators, some malware samples require minor modifications to bypass anti-analysis systems or to compensate for lost encryption keys. Examples of these modifications were provided for two malware families.

A review of the literature related to the creation of emulated C2 servers was undertaken. This review showed that the use of C2 server emulators and the automated generation of C2 server emulators is a new research topic with research in the early stages. Existing research provides the semi-automated generation of C2 server emulators based on individual samples.

The research in this chapter examines factors that need to be considered when creating an environment for the dynamic analysis of malware samples. Chapter 8 build on this with research, using dynamic analysis for the identification of ransomware. Two of the C2 emulators constructed in this chapter are used in Chapter 8 to allow CryptoWall and CryptoLocker malware to perform their full range of capabilities including encryption of user files in the VM.

Chapter 5

Evolved Similarity Techniques in Malware Analysis

This chapter presents a method for the identification of similar functions in pairs of malware variants. Malware authors are known to reuse existing code. This type of development process results in software evolution and a sequence of versions of a malware family containing functions that show a divergence from the initial version. The challenge in identifying evolved malware function pairs lies in identifying features that are relatively invariant across evolved code. The research in this chapter makes novel use of the function call graph in feature extraction and develops a method for the identification of evolved functions using ad-hoc function similarity comparison techniques. This work builds on the extraction of function similarity features from related parts of the call graph.

The contribution of this chapter has been published as a conference paper:

- P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Evolved Similarity Techniques in Malware Analysis," *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, ERA A, 2019.

2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering

Evolved Similarity Techniques in Malware Analysis

Paul Black* Iqbal Gondal† Peter Vamplew§ Arun Lakhotia‡
*paulblack@students.federation.edu.au, †iqbal.gondal@federation.edu.au, §p.vamplew@federation.edu.au, ‡arun@louisiana.edu

***Abstract*—Malware authors are known to reuse existing code, this development process results in software evolution and a sequence of versions of a malware family containing functions that show a divergence from the initial version. This paper proposes the term evolved similarity to account for this gradual divergence of similarity across the version history of a malware family. While existing techniques are able to match functions in different versions of malware, these techniques work best when the version changes are relatively small. This paper introduces the concept of evolved similarity and presents automated Evolved Similarity Techniques (EST). EST differs from existing malware function similarity techniques by focusing on the identification of significantly modified functions in adjacent malware versions and may also be used to identify function similarity in malware samples that differ by several versions. The challenge in identifying evolved malware function pairs lies in identifying features that are relatively invariant across evolved code. The research in this paper makes use of the function call graph to establish these features and then demonstrates the use of these techniques using Zeus malware.**

***Keywords*-malware evolution, malware similarity, Zeus, binary similarity**

I. Introduction

Malware authors are known to perform development based on existing code in order to maintain their software development efficiency [1], [2]. This development process results in a sequence of malware samples containing functions that gradually diverge in similarity from the initial version.

Evolved similarity describes the divergence of similarity across different versions of a malware family. While this paper does not compute evolutionary distance [3], it does make use of the concept of evolutionary distance to refer to the degree of difference between the functions in two versions of a malware program. This paper provides initial research in the development of Evolved Similarity Techniques (EST) that aim to maximise the evolutionary distance over which function similarity can be computed. To the best of our knowledge, this is the first time that malware function similarity research has considered developing methods with a goal of operating correctly over evolutionary distance.

New versions of a malware family may have functions added, functions may be removed, changes may occur due to compiler settings or software development. These changes are addressed by existing graph isomorphism or semantic similarity techniques [4], [5], [6], [7]. The performance of existing malware function similarity techniques decrease as the evolutionary distance between the malware samples increases.

Manual reverse engineering of the malware samples used in this research was performed to identify equivalent functions to allow assessment of experimental results. Given that this reverse engineering is used for evaluation purposes during the development of the research methods, this is a different situation from manual analysis where manual reverse engineering is required in each comparison.

This paper develops the concept of evolved similarity and provides a practical demonstration of the initial research of EST using two different versions of historic Zeus malware [8] samples. These Zeus samples were selected due to their version proximity (2.0.8.7 and 2.0.8.9) to the leaked Zeus source code (2.0.8.9) [9]. The Zeus malware family was selected due to the availability of samples, existing analysis, leaked source code and embedded version numbers in some variants. Automatic unpacking, disassembly and extraction of function semantics from the malware samples was performed using Cythereal MAGIC [10].

A. Evolved Similarity

The first situation where EST are useful arises when two adjacent or similar versions of malware are being compared. Here the aim is to identify those functions where software development has been performed, this allows the identification of functional changes in a new malware version. Identical functions, and functions that exhibit minor differences in code generation due to compiler settings are ignored.

Malware detection requires a basic level of analysis. When a new version of a malware family is identified, it may no longer be identified by existing detection techniques. Analysis of the new version may be necessary in order to determine why detection is failing.

A more detailed level of malware analysis is required to identify the command and response elements of the malware's network protocol. Analysis of new versions of malware is necessary in order to identify new commands and capabilities, this frequently requires manual analysis, which is time consuming, requires skilled analysts and does not scale well [11].

The second situation where EST are useful arises when two malware variants are being compared, for example Zeus V2 and Zeus P2P [12]. In this case, the older variant may have already been reverse engineered while the new variant has not been examined. The goal here is to identify previously analysed functions in the older variant that are present in a modified form in the newer variant. This allows knowledge of the purpose of these common functions to be read across from the old variant to the newer variant.

Significant software development may have been performed in a new variant, and common functions may have been refactored. One of the goals of EST is to indicate function similarity between refactored functions. The foundations of refactored similarity are developed in this research.

A challenge in developing EST lies in the fact that one of the functions being compared may have been modified in an arbitrary manner. In order to allow comparison in this situation, comparison features are taken from the target function and also from the call graph [13] of the target function.

B. *Contribution*

This paper makes the following contributions:

- Development of a technique for extracting invariant features associated with malware function pairs that differ due to software development.
- Development of a technique for identifying malware function pairs where software development has been performed.
- Creation of a collection of Interactive Disassembler (IDA) databases of Zeus malware samples with function labelling corresponding to the leaked Zeus source code.

The structure of this paper is as follows: the related work examines existing research, followed by research methodology including invariant features, function pair identification, analysis of results, and the conclusion.

II. Related Work

A. *Software Evolution*

Research into software evolution has largely focused on source code analysis in order to provide an understanding of program change over time [14]. Low level details of program evolution can be extracted from source code as a series of changes to a program's Abstract Syntax Tree (AST). The significance of program changes can be measured in terms of edit distance between AST states. Low level changes are aggregated to changes in program semantics, such as refactoring, addition or removal of features, or the correction of errors in the program. This aggregation of low-level changes supports reasoning about software evolution at different levels of abstraction and provides understanding of change in program semantics from a high-level view down to an implementation level [15]. Source code based software evolution approaches are not applicable to the majority of malware families due to the lack of availability of malware source code, although there are cases where malware source code has been made public [9], [16].

B. *Malware Lineage*

A common approach in malware research is to treat every packed sample as a distinct version [17]. The majority of malware samples are packed, the purpose of packing malware is to obfuscate the original malware and to hinder detection. While there are a large number of different packed malware samples, unpacking reveals a significantly smaller set of seminal malware families [17], [18]. Malware exhibits evolutionary relationship in an analogous manner to software programs in general. This evolutionary relationship is known as malware lineage. The research in [17] clusters malware samples by malware family, divides the samples of each malware family by version and then establishes the relationships between the versions and produces a lineage graph of the malware version lineage. Malware Analysis and Attribution Using Genetic Information (MAAGI) [11] determines malware similarity by clustering malware using family relationships, utilises sample temporal sequences to identify malware lineage, and finally identifies malware behaviours [8] and purpose.

C. *Inline Function Identification*

Inlined function and library identification [19] uses the Execution Flow Graph (EFG) that is a hybrid of control flow graph and data flow techniques. The EFGs are annotated with hashes derived from normalised instructions and operands. Instructions are normalised into the following categories: Data Transfer, Arithmetic Instruction, Logical Instruction, Control Transfer, String Instruction, and Default. Graph isomorphism is used to locate inlined compiler intrinsics and library functions in open source programs. While instruction normalisation may provide a simple technique for code comparison, problems may be encountered when instructions are used in multiple contexts. For example, a register may be zeroed by either of the following instructions, `mov eax,0` or by `sub eax,eax`. These alternate instructions may lead to the instruction zeroing a register being classified as either a data transfer instruction or as an arithmetic instruction. While instruction normalisation may be simple to implement, its limitation in dealing with alternative instruction usage limits its accuracy.

D. *VirusBattle*

BinJuice is a tool that extracts abstract semantics, called juice, from a program executable. Symbolic interpretation of the basic blocks of each function in the program is used to generate basic block semantics. The execution of instructions alters the contents of CPU registers and memory. These changes in CPU registers and memory can be represented semantically as a set of simultaneous equations. The code does not need to be executed in a physical CPU to create this representation of code execution; the effects of code execution can be generated using a technique known as symbolic execution. The aim of generating abstract code semantics is that any two equivalent code sequences can be represented by the same semantics. [7], [20].

VirusBattle [21] provides a facility for the analysis of malware samples that is built on BinJuice. VirusBattle was designed to be able to identify equivalent functions. The compilation process that generates equivalent functions will not be expected to use the same register names or the same memory addresses. To facilitate the identification of equivalent functions, VirusBattle replaces register names and literal constants with typed logic variables. This generalisation of semantics yields an abstract semantics that is resistant to code variation due to compiler settings. VirusBattle generates logic variables in a consistent manner, so that different register and memory selections will be reduced to identical logic variables. [6], [20].

VirusBattle unpacks the submitted malware sample and creates four different representations of the unpacked code. The representations are: disassembled code, generalised code (gen_code), semantics and generalised semantics (gen_semantics or juice). Generalisation of code and semantics is performed by replacing register names and memory addresses with symbolic values. The instruction `mov(eax, dptr('0x3004'))` would be represented in gen_code as `mov(A,B)`, in semantics as `A=pre(memdw('0x3004'))`, and in gen_semantics as `E=C`. Basic block similarity can be established by string comparison of the hash of the block's juice and does not require the use of a theorem prover, thereby obtaining a significant performance gain [20]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

E. BinDiff

BinDiff provides a graph-based analysis of the differences between two versions of the same program, it designed to determine the program changes introduced by security patches [6]. BinDiff uses program structure and syntactic features such as string references when calculating similarity [22]. BinDiff identifies the functions in the two programs and calculates the Control Flow Graph (CFG) [23] of the functions and uses graph isomorphism to identify function pairs. Metrics such as edge counts, derived from the CFG are used to estimate function similarity. BinDiff initially calculates unique function matches. Additional matches are then found by searching for more unique matches from unmatched neighbouring functions [24]. The BinDiff user interface displays a ranked list of matching function pairs, a similarity ratio and a confidence statistic. BinDiff displays the Control Flow Graph (CFG) of selected functions and identifies matching, differing and removed basic blocks.

BinDiff operates well when the programs being compared exhibit similar CFGs, however recompiling a program with a high optimisation setting changes the CFG sufficiently to reduce Bindiff's accuracy to 25 percent [25].

III. Research Methodology

The aim of this research is to determine the significant differences in function pairs of the same malware family. Two Zeus malware samples were used in this research. Sample 1 was Zeus version 2.0.8.7 with a MD5 hash of `8a7faa25f23a0e72a760075f08d22a91d2c85f57`. Sample 2 was Zeus version 2.0.8.9 with an MD5 hash of `21a6db13a23ae35b31c4977a129391da3eac1d2e`. The version of the Zeus samples was taken from the Report::AddBasicInfo function.

Unpacking, disassembly and function semantics of the malware samples were created using Cythereal MAGIC. Interactive Disassembler (IDA) databases were created from each unpacked sample. IDA identified 577 functions in sample 1 and 553 functions in sample 2. The number of exactly matching functions in the two samples was 483.

A manual comparison of the two malware samples against the leaked Zeus source code was performed in order to identify and label the function pairs. The function pairs were then examined in order to determine the functions that exhibited differences due to significant software development. This manual analysis was based on existing IDA databases and was performed in order to create a ground truth on which to assess the automatic methods. While it would have been possible to compile the leaked Zeus source code to give a Zeus sample containing symbolic information, this would only provide one of the two samples that are required for this research. Manual analysis would still be necessary to label the functions in the second Zeus sample and to compare the function pairs.

The functions in the two samples were examined on a pairwise basis to ensure consistent identification. Unidentified function pairs were labelled with unique names. It was noted that sample 1 included 11 Zeus functions for stealing email related information, while sample 2 did not contain these functions. Sample 2 was found to have the following inlined functions: `Str::FindCharA`, `Str::FindCharW`, and `UserHook::ClearInput`. These functions were not inlined in sample 1. The Zeus malware kit was sold by a marketing process which included different capabilities at different prices. As a result, the modules compiled into Zeus malware samples vary on a per sample basis.

The research process was as follows. The malware samples were processed using Cythereal MAGIC, which provided unpacked malware, disassembly and function semantics. Manual comparison of the two Zeus samples was performed to identify the corresponding function pairs in the two samples, and then to identify those function pairs which exhibited significant differences. The function pairs were manually assessed as showing no change, minor change or significant change. The criteria for manually identifying significant change was a difference in the number of API calls, a difference in the number of function calls, or the addition or removal of code. This criteria is intended to exclude minor changes due to different compiler settings.

Table III shows the function pairs in the two malware samples where manual analysis identified significant differences. The function name column shows the function name corresponding to the Zeus source code. The remaining two columns show the Virtual Address (VA) [26] of the functions in the two samples. These VA's are provided to assist other researchers who may seek to replicate this research.

A. Invariant Features

When a new malware version is identified, the updated malware may exhibit machine code modification due to modification of source code, updated libraries or altered compiler options. The difficulty here is that there are no restrictions on the modifications that can be performed in the malware code. For example, the set of API calls from a function could be selected as a feature, but the updated malware version may have a different number of API calls in the corresponding function.

In relation to a CFG, a function `d` dominates its call graph as function `d` must be called prior to the calling of any of the functions in its call graph [23]. The term ”dominator set” is defined as the unique set of all static function calls called by the target function or its associated callgraph.

In this research, features have been extracted from the target function's dominator set. This approach is based on the incremental nature of software (and malware) development. While a specific function may have been modified due to version update, it is unlikely that all of the functions in the static call graph will also have been modified. It is noted, that in the case where most of the functions in a new version of malware have been updated, then this technique will fail.

The following features were extracted from the functions of the original malware sample and the updated version.

- `F1`, the concatenated semantics of the target function dominator set.
- `F2`, the unique set of API calls made by the target function dominator set.
- The number of static calls `F3S`, API calls `F3A` and dynamic calls `F3D` made by the target function dominator set.

The Difflib SequenceMatcher ratio function [27] is used to calculate the similarity ratio `SimRatio` of `F1` for the function pair. SequenceMatcher uses the longest matching sub-sequence algorithm [28] to determine a similarity ratio between the sequences. SequenceMatcher is similar to but predates the Ratcliff/Obershelp Gestalt Pattern Matching algorithm [29].

In the following equations, subscripts o and u refer to the target function from the original malware and from the updated malware respectively. The ratio of matching API calls `ApiRatio` is calculated from F2 as follows:

$$ApiRatio = F2_o/F2_u$$

The ratio of the products of the function call counts `CM` is calculated as follows:

$$CM = (F3S_o * F3A_o * F3D_o)/(F3S_u * F3A_u * F3D_u)$$

B. Function Pair Identification

When the functions in original malware version are compared with the functions in the updated version the following relationships exist:

- The updated function is the same as the original function (R1).
- The updated function shows minor changes due to compiler settings with alternate register selection, instruction reordering or alternate instruction selection (R2).
- The original function is not present in the updated malware (R3).
- A new function has been added to the updated malware (R4).
- The original function shows modifications resulting from source code changes, but has not been refactored (R5).
- The original functionality is present, but has been refactored in the updated malware (R6).

The aim of this research is to identify function pairs exhibiting significant change, that is, relationships (R5) and (R6). Exact matches (R1) and minor changes (R2) are ignored. Functions which have been added (R4) to or excluded (R3) from, the updated malware do not form function pairs and are ignored. Minor changes (R2) are excluded by ignoring functions with matching basic block counts. Similarity of the remaining functions is assessed by checking if the features exceed a trigger value. The trigger values that gave the best performance were determined by trial and error and are shown in Table I. Future research will seek to improve the function pair identification method.

Feature Name	Trigger Value
SimRatio	0.08
ApiRatio	50
CM	60

TABLE I
FEATURE TRIGGER VALUES

A special case exists where the target function and its call graph do not contain any API functions. This is the case for string, compression and cryptographic functions. In this case, the features and trigger values are shown in Table II.

Feature Name	Trigger Value
SimRatio	0.50
CM	60

TABLE II
FEATURE TRIGGER VALUES WHEN NO API CALLS PRESENT

The pseudocode of the algorithm used to identify the functions exhibiting evolved similarity is shown in Figure 1.

C. Results

The program for identifying the function pairs showing evolved similarity identified 123 function pairs. Of these, 106 function pairs were false positives. The remaining 17 matching pairs contained 13 of the 19 manually assessed function pairs that show significant change. Two of the remaining 4 pairs were excluded due to equal basic block count, this points to the difficulty in exactly aligning manual and automated analysis. The program which implemented this algorithm was written in python and executed in approximately 44 seconds on a

```
get call graph for each function
get basic block semantics for each function’s call graph
get API call set for each function’s call graph
get fn call statistics for each function’s call graph
for fn1 in program1
  for fn2 in program2
    if sha1(fn1) == sha1(fn2)
      exclude fn1, fn2                #Exclude exact matches
    if sub(fn1 BBlock Count, fn2 BBlock Count)
             > 0 #Exclude close matches
      break
    if SeqMatch.Ratio(basic block semantics fn1, fn2)
             > BBlock Sim Threshold
      if API Call set matches(fn1, fn2)
             > API Threshold
        fn call statistic =
             #API calls * #static calls * #dynamic calls
      if fn call statistic(fn1, fn2) > Fn Call Threshold
        Evolved Similarity Match found
```

Fig. 1. Evolved Similarity Algorithm

Function Name	**2.0.8.7 VA**	**2.0.8.9 VA**
Core::DefaultModuleEntry	4112C3	410B00
Core::GenerateBotId	410D3C	410591
Core::InitHooks	410643	40FEE8
Core::RunAsBot	411413	410C3F
DynamicConfig::Download	404FFA	415CDA
DynamicConfig::GetCurrent	404DEA	415AEC
HttpGrabber::AnalizeRequestData	40F5A4	41731D
HttpGrabber::FreeRequestData	40FE58	417C65
HttpTools::CatExtraInfoFromUrlToUrlA	41BC44	40C81C
LocalConfig::GetCurrent	41182B	416835
Nspr4Hook::FillRequestData	40579D	40EE27
Nspr4Hook::HookerPrWrite	4061EC	40F8AF
Registry::GetValueAsBinaryEx	41AD3D	40BA42
SoftwareGrabber::FtpFlashFxp3Decrypt	414784	4111A3
SoftwareGrabber:: FtpFtpCommanderMarkStringEnd	415D5E	41276C
SoftwareGrabber::FtpWinScp	415B16	412AB3
UserHook::AddString	40738E	414181
WininetHook::FillRequestData	412A45	406414
WininetHook::OnHttpSendRequest	412B1F	40652F

TABLE III
MANUAL ANALYSIS OF ZEUS FUNCTIONS WITH SIGNIFICANT CHANGE

2.0.8.7 VA	**2.0.8.9 VA**	**Sim Ratio**	**API PCT**	**Calls Metric**	**TP/FN**
4112C3	410B00	0.16	35	6	FN
410D3C	410591				Decomp Err
410643	40FEE8	0.52	73	78	**TP**
411413	410C3F	0.06	60	95	FN
404FFA	415CDA	0.21	56	84	**TP**
404DEA	415AEC	0.23	100	100	**TP**
40F5A4	41731D	0.28	72	95	**TP**
41BC44	40C81C	0.59	100	81	**TP**
41182B	416835	0.19	100	100	**TP**
40579D	40EE27	0.28	93	73	**TP**
4061EC	40F8AF	0.19	68	89	FN
41AD3D	40BA42	0.58	80	75	**TP**
414784	4111A3	0.58		80	**TP**
415B16	412AB3	0.34	77	98	**TP**
40738E	414181	0.17	71	80	**TP**
412A45	406414	0.09	80	66	**TP**
412B1F	40652F	0.27	70	98	FN
41AD3D	40BA42	0.58	80	75	**TP**
40738E	414181	0.17	71	80	**TP**

TABLE IV
EXPERIMENTAL RESULTS FOR FUNCTION PAIRS MANUALLY IDENTIFIED AS SHOWING SIGNIFICANT CHANGE

3.4 GHz i7-3770 cpu. The experimental results are shown in Table IV for the 19 functions which were manually assessed as showing significant development.

The first two columns of Table IV give the VAs for the functions from sample 1 and sample 2 respectively. The third column shows the Sequence Matcher ratio for the concatenated semantics for the functions from the samples. The fourth column shows the percentage similarity of the unique API calls for the functions from the samples. The fifth column shows the percentage ratio of the function calls for the functions from the samples. The sixth column shows an assessment of whether the identified function pair represents a True Positive (TP) or a False Positive (FP).

The second row of Table IV corresponding to Zeus function `Core::GenerateBotId` is labelled as ”Decomp Err”. Investigation of this problem showed that this function was not decompiled in the results returned from Cythereal. The 13th row of this table contains a blank value for `ApiRatio`, this corresponds to Zeus Function `SoftwareGrabber::FtpFlashFxp3Decrypt`. This function and its corresponding call graph do not contain any API calls. The third row of this table corresponds to Zeus function `Core::RunAsBot`. The `SimRatio` for this function is 0.06, while the `ApiRatio` is 60 and the `CM` is 95. Examination of the call graph of this function shows that significant code refactoring has occurred, however the API similarity and function count similarity are good indicators of similarity. This case indicates that the use of ensemble methods in future research could be worth investigating [30].

IV. CONCLUSION

This paper examines the divergence of similarity between functions in an existing and a new malware version, and develops a method for identifying those function pairs that have been subject to software development. These functions correspond to the capabilities added in new malware versions.

In this research, identical functions and those exhibiting minor changes due to compiler differences, do not represent the addition of malware capability and are excluded, leaving those function pairs that exhibit differences resulting from software development.

This paper presents research that provides a technique for extracting invariant features from the call tree of malware functions. This technique is then used to identify function pairs that exhibit changes resulting from software development. To support this research, a collection of IDA databases of unpacked Zeus samples with function labelling corresponding to the leaked Zeus source code has been created.

Two Zeus versions were selected for this research due to their version proximity to the leaked Zeus source code. The functions in these malware samples were manually labelled with the function names from the source code. Corresponding function pairs were manually analysed in order to identify functions that had been changed in the new malware version.

Automated methods were developed through the identification of invariant features associated with the modified functions. These invariant features were created from the dominator set of functions from the call graph.

The malware samples used in this research contained 557 and 553 functions. Manual analysis of the labelled IDA database identified 17 functions that exhibited significant differences. The automated method in this research identified 13 of the 17 manually identified functions along with 106 false positives.

Based on the research in this paper, there is scope for the following future research:

- Development of a formal model of evolved similarity.
- Development of metrics to measure the degree of code evolution between functions.
- Evaluation of the performance of each of the comparison metrics and the identification of additional metrics with improved performance.
- The research presented in this paper is performed on a pair of closely related Zeus malware samples. This research could be extended using less closely related pairs of other malware families.

ACKNOWLEDGEMENT

The authors would like to thank Cythereal[1] for providing access to Cythereal MAGIC and to the malware dataset used in this research. This research was funded in part through the Internet Commerce Security Laboratory (ICSL), a joint venture between Westpac, IBM and Federation University Australia.

REFERENCES

[1] M. Alazab, "Profiling and classifying the behavior of malicious codes," *Journal of Systems and Software*, vol. 100, 2015.
[2] I. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *Computers & Security*, vol. 78, pp. 347–363, 2018.
[3] G. Wagener, A. Dulaunoy *et al.*, "Malware behaviour analysis," *Journal in computer virology*, vol. 4, no. 4, pp. 279–287, 2008.
[4] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *SSTIC*, vol. 5, no. 1, p. 3, 2005.
[5] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security*. Springer, 2008, pp. 238–255.
[6] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 5.
[7] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code re-use," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE, 2013, pp. 492–501.
[8] P. Black, I. Gondal, and R. Layton, "A survey of similarities in banking malware behaviours," *Computers & Security*, vol. 77, pp. 756–772, 2018.
[9] Zeus Author, "Zeus source code," 2011. [Online]. Available: https://github.com/Visgean/Zeus
[10] Cythereal Inc, "Cythereal MAGIC," 2018. [Online]. Available: https://www.cythereal.com
[11] A. Pfeffer, C. Call, J. Chamberlain, L. Kellogg, J. Ouellette, T. Patten, G. Zacharias, A. Lakhotia, S. Golconda, J. Bay *et al.*, "Malware analysis and attribution using genetic information," in *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on*. IEEE, 2012, pp. 39–45.

[1]Cythereal has licensed VirusBattle from the University of Louisiana at Lafayette

[12] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly resilient peer-to-peer botnets are here: An analysis of gameover zeus," in *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*. IEEE, 2013, pp. 116–123.
[13] K. Cooper and L. Torczon, *Engineering a compiler*. Elsevier, 2011.
[14] J. Dagit and M. Sottile, "Identifying change patterns in software history," *arXiv preprint arXiv:1307.1719*, 2013.
[15] R. Robbes, M. Lanza, and M. Lungu, "An approach to software evolution based on semantic change," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2007, pp. 27–41.
[16] G. Brindisi, "Malware source codes," 2018. [Online]. Available: https://github.com/gbrindisi/malware
[17] I. U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *arXiv preprint arXiv:1710.05202*, 2017.
[18] F. C. C. Osorio, H. Qiu, and A. Arrott, "Segmented sandboxing-a novel approach to malware polymorphism detection," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015, pp. 59–68.
[19] J. Qiu, X. Su, and P. Ma, "Using reduced execution flow graph to identify library functions in binary code," *IEEE Transactions on Software Engineering*, vol. 42, no. 2, pp. 187–202, 2016.
[20] A. Lakhotia and P. Black, "Mining malware secrets," in *Malicious and Unwanted Software (MALWARE), 2017 12th International Conference on*. IEEE, 2017, pp. 11–18.
[21] C. Miles, A. Lakhotia, C. LeDoux, A. Newsom, and V. Notani, "Virus-battle: State-of-the-art malware analysis for better cyber threat intelligence," in *Resilient Control Systems (ISRCS), 2014 7th International Symposium on*. IEEE, 2014, pp. 1–6.
[22] C. LeDoux, A. Lakhotia, C. Miles, V. Notani, and A. Pfeffer, "Functracker: Discovering shared code to aid malware forensics," in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2013.
[23] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.
[24] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.
[25] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security Symposium*, 2014.
[26] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.
[27] D. Hellmann, *The Python standard library by example*. Addison-Wesley Professional, 2011.
[28] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. IEEE, 2000, pp. 39–48.
[29] J. W. Ratcliff and D. E. Metzener, "Pattern-matching-the gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.
[30] E. Menahem, A. Shabtai, L. Rokach, and Y. Elovici, "Improving malware detection by applying multi-inducer ensemble," *Computational Statistics & Data Analysis*, vol. 53, no. 4, pp. 1483–1494, 2009.

Chapter 6

Identifying Cross-Version Function Similarity Using Contextual Features

The research in this chapter builds on the EST method from the previous chapter, and replaces the ad-hoc function similarity method with a SVM model, and expands the set of features used in the EST method. While other machine learning techniques may offer potentially higher efficiency, the performance of the SVM model was found to be adequate for research purposes. A function's context is a set of functions comprising the function itself, and all the program functions that may be executed when this function is called. Contextual features consist of data that is extracted from the functions contained in the function context. This chapter identifies function pairs in two programs using representative features based on both individual functions and function context. This technique presented in this chapter uses a SVM model to pre-filter function pairs and then applies an edit distance technique using function semantics to reduce false positives. A performance analysis of the algorithm used in this chapter is provided in chapter 7. The contributions of this chapter are presented as a research paper:

- P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Identifying Cross-Version Function Similarity Using Contextual Features", *IEEE TrustCom*, IEEE, ERA A, 2020 (Accepted).

6.1 Introduction

The identification of similar functions in malware assists analysis by supporting the exclusion of functions that have been previously analysed, enables faster identification of new variants, supports authorship attribution, and the analysis of malware phylogeny [13]. Similar functions in a pair of malware samples may be due to the samples being from the same malware family, the samples being produced by the same malware author, use of shared code, a common library, or a well-known malware technique [79].

The organisations performing malware analysis face several challenges. The first is the large number of packed malware samples that must be processed daily. Unpacking of these malware samples reveals the original malware program. A comparison of the unpacked malware programs allows the elimination of exact matches, thereby reducing the volume of new malware samples by several orders of magnitude [15, 80]. Clustering may be used to segregate the unique malware samples into malware variants [81]. Previously analysed variants are removed from the clustered samples using malware similarity, some of which use function comparison [82]. Function similarity can be used to exclude functions that are unchanged and allow analysis to focus on changed functions.

Software evolution occurs in most malware families as a result of malware development efforts. As new malware versions are released, the machine code in the malware functions diverges from that of previous versions. Modeling of the generation and evolution of malware has been performed [79]; however, matching compiled functions across multiple malware versions remains a significant research challenge.

This paper presents Cross Version Contextual Function Similarity (CVCFS), a 2-step method for the identification of similar pairs of functions in two variants of a program. In the first step, a Support Vector Machine (SVM) model is used to pre-filter candidate pairs from a set of all pairs of functions from the two variants. The innovation in this step lies in the use of features that stay reasonably invariant, even under refactoring and program evolution. These features termed – contextual features – capture the calling context of a function. In the second step, false positives from the first step are weeded out by using edit distance between over function semantics.

A case study is performed where function similarity features for training were extracted from Zeus version 2.0.8.7 and Zeus 2.0.8.9 malware samples. The compilation dates of these training samples differ by approximately one month, with minor variations between the functions. The malware samples were labelled with ground truth function matching labels that were obtained by manual reverse engineering. An SVM model was trained using the labelled features. Function pair prediction was tested using Zeus version 2.0.8.9 and 2.1.0.1 malware samples. The compilation dates of these malware samples differ by approximately six months with significant variation in some of the functions. A comparison of the SVM model results demonstrates the higher performance of contextual features. The SVM pre-filtering is followed by the use of an edit distance technique to filter false-positives. This resulted in a function pair identification accuracy of 85 percent, this was verified against the baseline function similarity obtained by manual reverse engineering.

This paper makes the following contribution:

- Introduces a new class of features called contextual features that strengthen features from individual functions. These are well suited for capturing similarity between malware samples exhibiting evolutionary changes.
- Cross Version Contextual Function Similarity (CVCFS) technique for finding similar functions in pairs of programs, malware, or benign.

The following additional contributions are provided:

- A curated dataset of pairs of matched functions in three versions of Zeus malware for use in future research.
- A set of three labelled IDA databases of Zeus malware versions 2.0.8.7, 2.0.8.9, and 2.1.0.1. [1].
- Creation of labelled Interactive Disassembler (IDA) databases of Zeus versions 2.0.8.7, 2.0.8.9, 2.1.0.1 malware samples.

The structure of this paper is as follows: Section II presents related work, Section III presents the research methodology, Section IV presents the empirical evaluation of results, and Section V presents the conclusion.

[1]The datasets related to this research are available online at http://federation.edu.au/icsl/evolvedsimilarity

6.2 Related Work

6.2.1 BinJuice Function Semantics

BinJuice computes abstract semantics for each basic block of all functions in a compiled program by a process of disassembly, control flow graph (CFG) generation, symbolic execution, algebraic simplification, and the computation of function semantics [49]. The aim of generating abstract semantics is to represent any two equivalent code sequences by the same semantics [54, 57].

Disassembly and CFG extraction are performed by existing tools. Basic block semantics are generated using symbolic interpretation. Symbolic execution does not involve execution on a physical processor; instead, the effects of the program instructions can be represented as a set of simultaneous equations. An algebraic simplifier provides simplification of the symbolic expressions resulting from the symbolic execution. The simplified symbolic expressions are mapped into a canonical form to ensure that sections of equivalent code having equivalent symbolic expressions are sorted in the same order. The juice operation addresses the problem of comparing sections of equivalent code containing different register allocations by renaming register names in the simplified symbolic expressions to consistent logical variables [49].

VirusBattle is built on BinJuice and is used to identify relationships between malware samples in a large database [56]. VirusBattle unpacks the submitted malware sample and creates four different representations of the unpacked code. These representations are disassembled code, generalised code, semantics, and generalised semantics (juice). VirusBattle identifies similar functions by comparing the hashes of the function's generalised semantics, this provides a significant performance gain compared with non-approximate methods, e.g., theorem solvers [57]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

6.2.2 Machine Learning In Software Similarity

The proliferation of IoT devices using open source code and a variety of CPU architectures has led to a research interest in the identification of known defects in the same source code compiled for several different CPU architectures. Examples of this research are provided by Gemini [59] and SAFE [60]. Early approaches to function binary similarity made use of graph isomorphism. These techniques were effective, but performance becomes an issue with increasing graph size. Neural networks have been used for cross-architecture bug-search as these techniques exhibit better performance characteristics than graph isomorphism.

Gemini performs cross-architecture, binary, software defect search using Control Flow Graph (CFG) features. Gemini uses a neural network to extract features from the CFG of the compiled functions [59].

Cross-architecture function similarity is also addressed by SAFE, which extracts the instruction sequence from compiled functions and models them as a natural language [60]. The interaction of the instruction sequence is captured using a Gated Recurrent Unit Neural Network (GRU-RNN). An attention mechanism is used to automatically focus on the instructions with the best performance for the identification of similar functions.

The techniques in the above research use neural networks to locate software vulnerabilities in the same version of code that has been compiled for different CPU architectures. These techniques are optimised for cross-architecture vulnerability identification within a program of the same software version. The techniques provided in this paper provide identification of similar functions in different versions of the same program compiled for a single CPU architecture.

6.3 Research Methodology

This research provides the CVCFS technique that is used to identify functions that were compiled from different versions of the same malware family. This technique uses an SVM model to pre-filter function pairs, and then uses an edit distance technique on the function semantics to filter and significantly reduce false-positives from the preliminary function pair identification.

Previous research using ad-hoc methods and hard-coded heuristics were used to identify function similarity [83]. CVCFS builds on this research by replacing the heuristics and ad-hoc methods with a Support Vector Machine (SVM) model.

SVM is a machine learning model for binary classification [84]. The SVM algorithm divides an n-dimensional feature space into two classes using a hyperplane. The CVCFS program extracts features from functions in the disassembly of two related programs; an SVM model is used to identify similar function pairs in the two programs.

The Function Similarity Ground Truth (FSGT) dataset contains data defining the pairing of the functions from the two programs being compared. The FSGT dataset contains a name that has been assigned to each function pair and the Relative Virtual Addresses (RVAs) of each function in the pair. The function names are used for researcher convenience and are not used in the similarity algorithms.

6.3.1 Function Context

Given a program p containing a set of functions F, the context *c(f)* of a specific function *f* is function *f*, plus the set of all non-API functions *f'* that can be reached by walking the call graph starting from function *f*.

- **Control Flow Edge** A control flow edge e represents the transfer of control that occurs when function x calls function y. A control flow edge e from function x to function y is represented by e = *(x,y)*.

- **Call Graph** A call graph is represented by the directed graph G = *(V,E)*, where V is the set of functions in a program, and E is the set of control flow edge transitions.

- **Walk** A walk w in G is a finite set of control flow edge transitions that occur as the result of a sequence of function calls from the execution of function *f*.

$$w(f) = \{e_0, e_1, ..., e_{\mathrm{i}}\} \tag{6.3.1}$$

- **Path** A path w in G is the set of vertices v traversed due to a walk in a call graph G, this path represents the set of functions *f'* called by the execution of function *f*.

$$w(f) = \{v_0, v_1, ..., v_{\mathrm{i\text{-}1}}\} \tag{6.3.2}$$

- **Function Context** The context *c(f)* of function *f* is the set of all functions *f'* that can be reached by walking all possible paths in G starting from the vertex representing function *f*. Recursion and call graph loops require limiting the walk to paths not previously walked.

$$c(f) = \{f' : f' \epsilon \, \forall \, p(f)\} \tag{6.3.3}$$

6.3.2 Local Features

The CVCFS technique makes use of local features taken only from function *f*, and contextual features taken from the context *c(f)* of the function. The local features consist of the following:

- Set of API calls,
- Set of constants,
- Stack size,
- Function callers count.
- Basic block count,

Set of API calls: The system programming interface for the Windows operating system is provided by the Windows Application Programming Interface (API) [85]. This API provides a dynamic function call interface for Windows system services. Windows programs use the portable executable (PE) format. In the case where a call to an API results in calls subsequent API's, only the first API call is recorded. Let *AL(f,p)* be the set of API functions called by function f in program p.

$$AL(f,p) = \{a_0,\ a_1,\ \dots\ ,a_n\}. \tag{6.3.4}$$

Set of constants: The goal in extracting a set of constants is to extract invariant numerical constants from the operands of instructions in functions. Call and jump instructions were excluded because they have operands that contain program and stack addresses that are not invariant. Let *CL(f,p)* be the set of constants that are not program or stack addresses contained in function f of program p.

$$CL(f,p) = \{c_0, c_1, ..., c_{\mathrm{m}}\} \tag{6.3.5}$$

Stack size: Let *SL(f,p)* be the stack size s_0 of function f in program p.

$$SL(f,p) = s_0 \tag{6.3.6}$$

Function callers count: Let *FL* be the count of calls made to function f in program p.

$$FL(f,p) = |\{c_0, c_1, ..., c_{\mathrm{n}}\}| \tag{6.3.7}$$

Basic block count: A basic block is defined as the maximal sequence of consecutive instructions that begin execution at the first instruction and when the block is executed, all instructions in the basic block are executed sequentially without halting or branching, except for the last instruction in the block [86]. Let *BL* be the count of basic blocks in function *f*.

$$BL(f,p) = \{b_0\} \tag{6.3.8}$$

6.3.3 Local Feature Ratios

The CVCFS system calculates feature ratios using the Cartesian product of all functions in program *p1* and all functions in program *p2*. It is noted that function similarity is commutative, and the same function pairs will be identified by comparing programs *p1*, *p2* as would be identified by comparing program *p2*, *p1*. Let *F(p)* be the set of all functions in program *p*.

$$F(p) = \{f_0, f_1, ..., f_{\mathrm{l}}\} \tag{6.3.9}$$

The set of function pairs *FP(p1,p2)* of programs *p1* and *p2* is defined as follows:

$$FP(p1,p2) = F(p1) \times F(p2) \tag{6.3.10}$$

Each element of the Cartesian product *FP* is a function pair *fp* consisting of one function *f1* from program *p1* and one function *f2* from program *p2*.

$$fp = (f1, f2) \tag{6.3.11}$$

Local API Ratio: Let *ALE1* and *ALE2* be the sets of API calls extracted from each of the functions in function pair *fp*. Let local API ratio ARL, be the ratio of the magnitude of the intersection of *ALE1* and *ALE2* to the larger of the magnitude of *ALE1* and *ALE2*.

$$ARL = len(ALE_1 \cap ALE_2)/max(len(ALE_1), len(ALE_2)) \tag{6.3.12}$$

Local Constants Ratio: Let *CLE1* and *CLE2* be the sets of constants extracted from each of the functions in function pair *fp*. Thus *CL1 = CLE(fp[0])* and *CL2 = CLE(fp[1])*. Let local constants ratio LCR be the ratio of the magnitude of the intersection of *CLE1* and *CLE2* to the larger of the magnitude of *CLE1* and *CLE2* .

$$CRL = len(CLE_1 \cap CLE_2)/max(len(CLE_1), len(CLE_2)) \tag{6.3.13}$$

Local Stack Ratio: Let *SLE1* and *SLE2* be the stack sizes extracted from each of the functions in function pair *fp*. Let local stack ratio *LSR* be the ratio of the magnitude of the absolute value of the difference between *SLE1* and *SLE2* to the larger of the magnitude of *SLE1* and *SLE2* .

$$SRL = abs(SLE_1 - SLE_2)/max(len(SLE_1), len(SLE_2)) \tag{6.3.14}$$

Callers Ratio: Let *FCE1* and *FCE2* be the function callers counts extracted from each of the functions in function pair *fp*. Let *FCD* be the absolute value difference between *FCE1* and *FCE2*. Then *FCD* = abs(FCE$_1$ - FCE$_2$). Let callers ratio *CR*, be the ratio of *FCD* to the larger of *FCE1* and *FCE2*.

$$CR = FCD/max(FCE_1, FCE_2) \tag{6.3.15}$$

Blocks Ratio: Let *BLE1* and *BCE2* be the basic block counts extracted from each of the functions in function pair *fp*. Let blocks ratio *BR* be the ratio of BLE_1 and BLE_2.

$$BR = min(BLE_1, BLE_2)/max(BLE_1, BLE_2) \tag{6.3.16}$$

6.3.4 Contextual Features

The strength of function similarity features can be improved by extracting the feature across all of the functions contained within the context of the function under consideration. The contextual features consist of the following:

- Contextual set of API calls,
- Contextual set of constants,
- Contextual stack size,
- Contextual return count,
- Contextual function calls count.

Contextual set of API calls: Let AC be the set of API calls made from the context *c(f)* of function *f*.

Contextual set of constants: Let *CC* be the set of constants from the context *c(f)* of function *f*.

Contextual stack size: Let *SC* be the sum of stack sizes from the context *c(f)* of function *f*.

Contextual return count: Let *RC* be the count of return instructions from the context *c(f)* of function *f*.

Contextual function calls count: Let *CS* be the count of call instructions from the context *c(f)* of function *f*.

6.3.5 Contextual Feature Ratios

The contextual feature ratios consist of the following:

Contextual API Ratio: Let AC_1 and AC_2 be the set of API calls made from the context of each of the functions in function pair *fp*. Let the contextual

API ratio *ACR*, be the ratio of the magnitude of the intersection of AC_1 and AC_2 to the larger of the magnitude of AC_1 and AC_2.

$$ACR\ = len(AC_1\ \cap\ AC_2)/max(len(AC_1), len(AC_2)) \tag{6.3.17}$$

Contextual Constants Ratio: Let CC_1 and CC_2 be the set of constants from the context of each of the functions in function pair *fp*. Let the contextual constants ratio *CCR* be the ratio of the magnitude of the intersection of CC_1 and CC_2 to the larger of the magnitude of CC_1 and CC_2 .

$$CCR = len(CC_1\ \cap\ CC_2)/max(len(CC_1), len(CC_2)) \tag{6.3.18}$$

Contextual Stack Ratio: Let SC_1 and SC_2 be the sum of the stack sizes from the context of each of the functions in function pair *fp*. Let the contextual stack ratio *SCR* be the ratio of the magnitude of the intersection of SC_1 and SC_2 to the larger of the magnitude of SC_1 and SC_2 .

$$SCR = len(SC_1\ \cap\ SC_2)/max(len(SC_1), len(SC_2)) \tag{6.3.19}$$

Contextual Returns Ratio: Let RC_1 and RC_2 be the count of return instructions from the context of each of the functions in function pair *fp*. Let the contextual returns ratio *RCR*, be the ratio of absolute value difference between RC_1 and RC_2 to the larger of RC_1 and RC_2.

$$RCR = abs(RC_1 - RC_2)/max(len(RC_1), len(RC_2)) \tag{6.3.20}$$

Contextual Calls Ratio: Let FC_1 and FC_2 be the count of call instructions from the context of each of the functions in function pair *fp*. Let the callers ratio *FCR*, be the ratio of absolute value difference between FC_1 and FC_2 to the larger of FC_1 and FC_2.

$$FCR = abs(FC_1 - FC_2)/max(FC_1, FC_2) \tag{6.3.21}$$

6.3.6 Edit Distance Filtering

The function similarity results obtained by the SVM model developed in an earlier section included a large number of false-positive results. Experimentation with the SVM model was performed but the false-positive results remained a problem. Existing research [10, 59] uses machine learning as the first stage pre-filter in identifying similar functions. To overcome the large number of false-positives, a decision was taken to use the SVM model as a pre-filter, and to add an edit distance metric using the BinJuice generalised semantics to filter out false-positives.

Although it would be possible to remove the pre-filtering step and to solely use the graph edit distance for the identification of function pairs, this would not be feasible due to the significant execution time of the edit distance calculation. In the research in this paper, the SVM pre-filter ran in approximately two minutes, while the run time for the edit distance filtering could take as long as 12 hours.

BinJuice function semantics contains four levels of abstraction, where the most abstract form of the function semantics is the Generalised Semantics. The Levenshtein edit distance [87] of the Generalised Semantics of each function was calculated. Edit distance increases with function size and cannot be used directly to identify matching function pairs. The edit distance was normalised by dividing the edit distance by the function basic block count to give a Normalised Edit Distance per Basic Block (NEDBB) metric The NEDBB metric was used to identify matching function pairs from the pre-filtered matches.

6.3.7 CVCFS Algorithm

The algorithm used to extract the function context of a specified function is shown in Figure 6.1. The algorithm for feature extraction is shown in Figure 6.2. The algorithm used to perform edit distance filtering of the pre-filtered function pairs is shown in Figure 6.3. In the edit distance filtering algorithm, the `filtervalue` is a hardcoded threshold that is tested against the NEDBB value in order to select the pre-filtered function pairs.

```
  Create empty function context
  Create empty visited list
  From the disassembly of function f
    Add static function calls from f to function context
    For each function in function context
      If function not in visited list
        Add function to visited list
        Recursively get new static functions called by function
        Add new static functions to function context
  Add f to function context
```

Figure 6.1: Function Context Extraction Algorithm

```
p1 = baseline version of program
p2 = updated version of program

  for each function in p1
    extract function context for function

  for each function in p2
    extract function context for function

  for each function in p1
    calculate individual features for function
    calculate contextual features for function

  for each function in p2
    calculate individual features for function
    calculate contextual features for function

  for each f1 in p1
    for each f2 in p2
      calculate locate feature ratios
      calculate contextual feature ratios
      if f1, f2 in FSGT dataset
        set label = "1"
      else
        set label = "0"

  rva1 = rva(f1)
  rva2 = rva(f2)
  features = map(rva1, rva2, ARL, ARD, FRD,
                 RRD, CR, LCR, CRD, LSR, LSD, LBR)
```

Figure 6.2: Feature Extraction Algorithm

```
let ml_predicted = all fn pairs predicted match
for each fn_pair in ml_predicted
  let sem1 = semantics of fn_pair[1]
  let sem2 = semantics of fn_pair[2]
  let ed = nktk.edit_distance(sem1, sem2)
  let bc1 = blocks count fn_pair[1]
  let ed_per_block = ed / bc1
  if ed_per_block < filter value
    match = True
  else
    match = False
```

Figure 6.3: Edit Distance Filtering Algorithm

6.4 Empirical Evaluation

The malware samples used in this paper are shown in Table 6.1. The unpacked samples were provided by Cythereal and were disassembled using the Interactive Disassembler (IDA). The linker date in the Portable Executable (PE) header is used to indicate the time the malware samples were created, although the linker date can be modified by the malware author, there are no inconsistencies that suggest this time has been modified. The linker dates indicate that sample 2 was produced approximately one month after sample 1, and sample 3 was produced approximately six months after sample 1.

The malware samples were submitted for Cythereal processing. The output from the Cythereal processing is the unpacked malware sample and a dataset containing the disassembly and semantics of the unpacked malware sample [88].

Sample SHA1 Hash	**Version**	**Date**
8a7faa25f23a0e72a760075f08d22a91d2c85f57	2.0.8.7	2010-09-14
706bf4dcf01b8eceedf6d05cf5b55a27e4ff8ef0	2.0.8.9	2010-10-15
30c6bb2328299a252436d2a3190f06a6f04f7e3f	2.1.0.1	2011-03-24

Table 6.1: Zeus Sample Details

Tool	**Version**	**Count**	**Version**	**Count**	**Match Count**
IDA	2.0.8.7	577	2.0.8.9	553	549
Cythereal	2.0.8.7	577	2.0.8.9	553	549
IDA	2.0.8.9	553	2.1.0.1	601	539
Cythereal	2.0.8.9	539	2.1.0.1	601	517

Table 6.2: Zeus Function Count and Manual Match Count

The FSGT dataset identifies the function pairs, function names, and function RVAs for each malware sample used in this research. Manual analysis using IDA, the unpacked malware samples, and the leaked Zeus source code was performed in order to identify the function pairs for the FSGT dataset. Function identification was performed based on the API calls, constants, and CFG structure. Function names from the Zeus source code provide a convenient identification for researchers but are not used by the research code. Function RVAs are used to identify functions in the Cythereal semantics and

```
push(ebp)
mov(ebp,esp)
sub(esp,SL)
```

Figure 6.4: Function Prologue

in the research code. The results of the function name labelling are shown in Table 6.2. IDA identified 553 functions in sample 3 while Cythereal MAGIC could only identify 539 functions in sample 3. This discrepancy may be an artifact of Cythereal processing.

6.4.1 Features

To calculate the local and contextual constant features, constants were extracted from `mov`, `push`, `add`, `cmp`, and `sub` instructions. Ad-hoc analysis showed that these instructions contained a significant proportion of invariant operands. Program and stack addresses were further filtered by excluding values greater than the program base address.

The stack size is taken from the function prologue when it is present; otherwise, it is zero. The stack size SL is taken from the `sub` instruction in the function prologue shown in Figure 6.4. It is noted that some compilers may not use the same idiom for their function prologue.

As this research uses static analysis, the non-API function call counts used in this research are a count of static function calls.

6.4.2 SMOTE Oversampling

The function similarity training dataset was imbalanced due to the use of the Cartesian product comparison, which resulted in an unstable performance of the SVM model. Assume that the two versions of the same program are being compared, and each program contains 500 functions. The maximum number of matching function pairs is 500. The number of function pairs generated by the Cartesian product is 250,000, and the minimum number of non-matching function pairs is 249,500. The use of a Cartesian product in the generation of training features inherently leads to an imbalanced dataset. The performance of the SVM model was improved with the use of Synthetic Minority Oversampling Technique (SMOTE) [89] to rebalance the training dataset.

6.4.3 SVM Model Training

In this paper, sample 1 and sample 2 were used for training as these two Zeus samples are similar but exhibit a number of minor differences. Function similarity features for training were calculated using the Cartesian product of all functions in samples 1 and 2. These features were labelled as matching or not matching using the FSGT dataset. All possible feature combinations were used to train a series of SVM models in order to identify the best performing feature combinations. As 10 features were used, exhaustive testing required 1023 tests to be performed. The features in each of these tests were assigned a binary identifier, e.g., the first feature is identified as `0000000001`, the second feature was identified as `0000000010`. The use of binary identifiers allowed the numbering of individual tests; these feature identifiers are shown in Table 6.3.

6.4.4 Pre-Filtering

The functions in samples 1 and 3 exhibit more differences due to software development than the training dataset. A testing set of function similarity features were created using the Cartesian product of all functions in samples 1 and 3. The previously generated SVM models were used to predict the matching function pairs from the testing feature set. The results of this prediction were evaluated using the FSGT dataset.

Feat #	**Vector**	**Description**
1	0000000001	Basic Block Ratio
2	0000000010	Contextual Stack Ratio
4	0000000100	Local Stack Ratio
8	0000001000	Contextual Constants Ratio
16	0000010000	Local Constants Ratio
32	0000100000	Contextual Callers Ratio
64	0001000000	Contextual Returns Ratio
128	0010000000	Calls Ratio
256	0100000000	Local API Ratio
512	1000000000	Contextual API Ratio

Table 6.3: Numbering For Feature Combination Tests

The performance of each individual feature was assessed by performing function similarity classification using SVM models trained for each individual feature. The results of this evaluation are shown in Table 6.4. Although the prediction of the SVM model was reasonable, the recall performance was not good resulting in a significant false-positive count. A number of features were tested in an effort to reduce the false-positive count, ultimately it was decided to use the SVM model as a pre-filter of function pairs.

The feature combinations which provided the best performance are shown in Table 6.5. The function pair prediction results in this research vary from run to run due to the stochastic nature of machine learning.

The F-measure (F1) [90] defined in equation 6.4.1 is used to assess the precision and recall of results.

$$F1 = 2 * (Precision * Recall) / (Precision + Recall) \tag{6.4.1}$$

Referring to the F-measure (F1) in Table 6.5, the best performing feature combination was test 611 with an F-measure of 0.07. For comparison, a random classifier would result in a low F-measure due to the unbalanced distribution of the classes within these datasets (Section 6.4.2). If we consider sample 3, a random classifier would classify half of the 539 matching function pairs as matching and half as non-matching, and would similarly split the 323400 non-matching function pairs between these two classes. This gives a precision score of 269.5 / (269.5 + 161700)=0.0017, and a recall of 269.5/539=0.5. Combining these gives an F-measure of 0.0034. The F-measures obtained by this SVM model are above those that would be expected using random classification. Feature combination `611` uses the Contextual API ratio, the Contextual Returns Ratio, the Contextual Callers Ratio, the Contextual Stack Ratio, and the Basic Blocks Ratio, as shown in Table 6.5.

6.4.5 Feature Performance

The relative performance of individual features can be assessed from Table 6.4 or it can be assessed from a count of those features present in the best performing feature combinations shown in Table 6.5. Table 6.6 summarises the performance of each of the features, the ”Ind Rank” column ranks individual feature performance based on the F-Measure with higher numbers indicating better performance. From this ranking, the Local API Ratio outranked the

Test	Vector	TP	FP	FN	Pr	Rc	F1
1	00000000001	500	64486	17	0.01	0.97	0.02
2	00000000010	280	19528	237	0.01	0.54	0.03
4	00000000100	265	20460	252	0.01	0.51	0.02
8	00000001000	479	10338	38	0.04	0.93	0.08
16	0000010000	460	18767	57	0.02	0.89	0.05
32	0000100000	378	69778	139	0.01	0.73	0.01
64	0001000000	490	36333	27	0.01	0.95	0.03
128	0010000000	444	65050	73	0.01	0.86	0.01
256	0100000000	279	746	238	0.27	0.54	0.36
512	1000000000	408	6565	109	0.06	0.79	0.11

Table 6.4: Individual Feature Performance

Contextual API Ratio, the Contextual Constants Ratio outranked the Local Constants Ratio, and the Contextual Stack Ratio outranked the Local Stack Ratio.

Feature performance was also assessed by observing the number of times the feature was present in the test runs of the Highest Performing Feature combinations in Table 6.5. This is shown in Table 6.6 as "TFP Count" (Top Performing Feature Count), this value is used to create a rank as shown in the "TPF Rank" column. In this ranking, the Contextual API Ratio outranks the Local API Ratio, and the Contextual Stack Ratio outranks the Local Stack Ratio, the Contextual Returns Ratio performed well with a ranking of 5. The features based on constants did not rank highly. The Basic Blocks Ratio ranked equally with the Contextual Stack Ratio.

6.4.6 Edit Distance Filtering

An edit distance metric was used to filter the pre-filtered function pair predictions to reduce the false-positive count from the SVM pre-filtering. The best performing pre-filter feature combination `611` from Table 6.5 was used in this edit distance filtering experiment. The results in Table 6.7 show the performance variation due to the use of different values of the Edit Metric Filter (`EMF`). This shows that two candidates for best performance are obtained with `EMF` values of 6 and 7 corresponding to an F-measure of 0.77. `EMF` value 7 was selected due to the higher true positive count.

Test	Vector	TP	FP	FN	Pr	Rc	F1
39	0000100111	510	66327	7	0.01	0.99	0.02
423	0110100111	509	50222	13	0.01	0.98	0.02
475	0111011011	511	25759	6	0.02	0.99	0.04
513	1000000001	510	21120	7	0.02	0.99	0.05
577	1001000001	509	41955	8	0.01	0.98	0.02
579	1001000011	509	21816	8	0.02	0.98	0.04
611	1001100011	509	12661	8	0.04	0.98	0.07
614	1001100110	509	42280	8	0.01	0.98	0.02
643	1010000011	511	42743	6	0.01	0.99	0.02
711	1011000111	509	26980	8	0.02	0.99	0.04
742	1011100110	511	54320	6	0.01	0.99	0.02
769	1100000001	512	69733	5	0.01	0.99	0.01
771	1100000011	511	34024	6	0.01	0.99	0.03
775	1100000111	512	63062	5	0.01	0.99	0.02
838	1101000110	509	32581	8	0.02	0.98	0.03
865	1101100001	511	31977	6	0.02	0.99	0.03
962	1111000010	509	45490	8	0.01	0.98	0.02

Table 6.5: Highest Performing Feature Combinations

In the results shown in Table 6.8, the best performing EMF value of 7 was used to reject false-positive predictions. The "ML" identifier in the operation ("OP") column denotes the results from the SVM pre-filtering stage. The "EM" identifier indicates the final results obtained using the edit distance filtering. Referring to Table 6.8, the highest F-measure (F1) value occurs with Test 614 with a value of 0.77, and 441 function pairs correctly identified. This corresponds to a function pair identification accuracy of 85 percent.

6.4.7 Future Work

Work presented in this paper can be extended as follows:

- Extend the concept of function context and its use in program analysis,
- Investigate the performance variation in existing function similarity programs over increasing evolutionary distance,
- Generalize the techniques used in this paper for identifying function similarity,

Feature	Ind Rank	TPF Count	TPF Rank
Basic Blocks Ratio	2	13	6
Contextual Stack Ratio	3	13	6
Local Stack Ratio	2	7	3
Contextual Constants Ratio	5	1	1
Local Constants Ratio	4	1	1
Contextual Callers Ratio	1	6	2
Contextual Returns Ratio	3	10	5
Calls Ratio	1	6	2
Local API Ratio	7	8	4
Contextual API Ratio	6	14	7

Table 6.6: Ranking of Feature Performance

Test	EMF	OP	TP	FP	FN	Pr	Rc	F1
611		”ML”	508	22248	9	0.02	0.98	0.04
611	5	”EM”	423	172	94	0.71	0.82	0.76
611	6	”EM”	438	178	79	0.71	0.85	0.77
611	7	”EM”	441	184	76	0.71	0.85	0.77
611	8	”EM”	444	214	73	0.67	0.86	0.76
611	9	”EM”	452	243	65	0.65	0.87	0.75
611	10	”EM”	458	279	59	0.62	0.89	0.73
611	11	”EM”	464	325	53	0.59	0.90	0.71
611	12	”EM”	469	395	48	0.54	0.91	0.68

Table 6.7: Edit Metric Filter Performance

- Automatically create the FSGT dataset,
- Research function similarity in programs that have been subject to significant software development.

6.5 Conclusion

The research in this paper introduces the concept of contextual features and provides methods for their calculation. A combination of individual and contextual features are used in the CVCFS technique for the identification of similar functions in two related programs. This paper demonstrates that contextual features provide improved performance compared to the corresponding features extracted from individual functions.

Test	Op	TP	FP	FN	Pr	Rc	F1
39	ML	505	41721	12	0.01	0.98	0.02
39	EM	443	231	74	0.66	0.86	0.74
423	ML	487	42284	30	0.01	0.94	0.02
423	EM	422	164	95	0.72	0.82	0.77
475	ML	496	7353	21	0.06	0.96	0.12
475	EM	429	167	88	0.72	0.83	0.77
513	ML	510	46724	7	0.01	0.99	0.02
513	EM	442	233	75	0.65	0.85	0.74
577	ML	504	14513	13	0.03	0.97	0.06
577	EM	440	183	77	0.71	0.85	0.77
579	ML	509	11928	8	0.04	0.98	0.08
579	EM	441	182	76	0.71	0.85	0.77
611	ML	509	24339	8	0.02	0.98	0.04
611	EM	441	185	76	0.70	0.85	0.77
614	ML	508	33414	9	0.01	0.98	0.03
614	EM	441	175	76	0.72	0.85	0.78
643	ML	423	6536	94	0.06	0.82	0.11
643	EM	360	146	157	0.71	0.70	0.70
711	ML	497	18675	20	0.03	0.96	0.05
711	EM	430	200	87	0.68	0.87	0.76
742	ML	475	15504	42	0.03	0.92	0.06
742	EM	413	154	104	0.73	0.80	0.76
769	ML	412	7199	105	0.05	0.80	0.10
769	EM	345	157	172	0.69	0.67	0.68
771	ML	428	11205	89	0.04	0.83	0.07
771	EM	360	156	157	0.70	0.70	0.70
775	ML	508	29646	9	0.02	0.98	0.03
775	EM	442	229	75	0.66	0.85	0.74
838	ML	507	34360	10	0.01	0.98	0.04
838	EM	440	192	77	0.70	0.85	0.77
865	ML	511	27297	6	0.02	0.99	0.04
865	EM	441	186	76	0.70	0.85	0.77
962	ML	456	16292	61	0.03	0.88	0.05
962	EM	413	173	104	0.70	0.80	0.75

Table 6.8: Results Following Edit Metric Filtering

The technique for the generation of contextual features involves summing features extracted from the context of each function. This technique of strengthening features by summing over the function context is generally applicable to a wide range of machine learning function similarity research.

The CVCFS function similarity technique is presented using a case study; however, this technique is generally applicable to identifying similar functions in both malware and benign programs. A comparison of the effectiveness of the local features and the contextual features was performed using a ranking based on feature performance and of the frequency of features in the highest performing feature combinations. This ranking shows that contextual features outperform local features in four out of six cases.

An edit distance technique was used to filter the false-positives from the machine learning results and a maximum accuracy of 85 percent identification of true function pairs was achieved with an F-measure value of 0.77.

Chapter 7

Function Similarity Using Family Context

The technique presented in this chapter builds on the CVCFS research from the previous chapter, using feature engineering methods to obtain a substantial improvement in the accuracy of identifying similar functions in two malware variants. In this research, the definition of function context is restricted to the selection of specific sets of features not just from the function itself, but also, from a limited number of other functions with which it has a caller and callee relationship. The encoding of numeric features in this research has been improved. These improvements to the definition of function context and the improved numeric feature encoding result in a substantial improvement in the accuracy of function similarity identification, and significantly decrease the false positive rate, obviating the need for a separate pass for cleaning false positives. This research uses a Zeus Support Vector Machine (SVM) model and applies it to the features extracted from a pair of ISFB malware variants. The successful prediction of function similarity in an unrelated pair of malware variants indicates that the SVM model is successfully abstracting function similarity features.

The contributions of this chapter have been published as:

- P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Function Similarity Using Family Context," *Electronics Journal*, Impact factor 2.412, 2020.

electronics

Article

Function Similarity Using Family Context

Paul Black [1,*], Iqbal Gondal [1], Peter Vamplew [2] and Arun Lakhotia [3]

1 Internet Commerce Security Laboratory (ICSL), Federation University, Ballarat 3353, Australia; iqbal.gondal@federation.edu.au
2 School of Engineering, Information Technology and Physical Science, Federation University, Ballarat 3353, Australia; p.vamplew@federation.edu.au
3 School of Computing and Informatics, University of Louisiana at Lafayette, Lafayette, LA 70504, USA; arun@louisiana.edu
* Correspondence: p.black@federation.edu.au

Received: 4 June 2020; Accepted: 13 July 2020; Published: 17 July 2020

Abstract: Finding changed and similar functions between a pair of binaries is an important problem in malware attribution and for the identification of new malware capabilities. This paper presents a new technique called Function Similarity using Family Context (FSFC) for this problem. FSFC trains a Support Vector Machine (SVM) model using pairs of similar functions from two program variants. This method improves upon previous research called Cross Version Contextual Function Similarity (CVCFS) representing a function using features extracted not just from the function itself, but also, from other functions with which it has a caller and callee relationship. We present the results of an initial experiment that shows that the use of additional features from the context of a function significantly decreases the false positive rate, obviating the need for a separate pass for cleaning false positives. The more surprising and unexpected finding is that the SVM model produced by FSFC can abstract function similarity features from one pair of program variants to find similar functions in an unrelated pair of program variants. If validated by a larger study, this new property leads to the possibility of creating generic similar function classifiers that can be packaged and distributed in reverse engineering tools such as IDA Pro and Ghidra.

Keywords: malware similarity; malware evolution; function similarity; binary similarity; Zeus malware; ISFB malware; machine learning

1. Introduction

This paper provides a technique called Function Similarity using Family Context (FSFC). This is a 1-step method using a Support Vector Machine (SVM) model with a low false positive rate for identifying similar pairs of functions from two variants of a program. The primary innovation is to demonstrate that our SVM model abstracts function similarity features in function pairs from two program variants to find similar functions in an unrelated pair of program variants. A second innovation is a new technique that extracts features from each function under consideration as well as specific related functions. These contextual features strengthen the function similarity results and provide a substantial performance improvement over and above training with features taken from individual functions. FSFC is a one-step method for function similarity that performs well without pre-filtering and provides a low false positive rate. FSFC is built on previous research called Cross Version Contextual Function Similarity (CVCFS) [1].

Function similarity techniques are used for malware triage [2], program patch analysis [3], identification of library functions containing known bugs [4,5], malware authorship analysis [6], identification of similar function pairs in detailed malware analysis [7], and for plagiarism analysis [8].

Organisations performing anti-virus or threat intelligence work process large daily feeds of malware samples. Due to packing, these malware samples initially appear to be unique. However, after unpacking malware sample volumes may be reduced by orders of magnitude through the use of malware similarity techniques to reduce the malware feed to a smaller set of unique malware payloads [9]. Function level similarity can then be performed on the unique malware payloads for malware family identification.

Software vendors provide program patches for maintenance and security purposes. High-level details describing the program patches may be provided, but this level of detail may not be sufficient for some organisations. Program patch analysis uses function similarity for the identification of updated functions. This is the first step in a reverse engineering process to identify the vulnerabilities that have been patched.

Internet of Things (IoT) devices, including routers and consumer electronics, utilize open-source software and a wide range of Central Processing Unit (CPU) architectures. When software vulnerabilities are identified in open source libraries, a wide range of IoT devices become vulnerable. Function similarity techniques are used for cross-architecture bug search purposes to search IoT firmware for vulnerable functions.

Law enforcement anti-malware efforts prioritize the identification of malware authors. Function similarity techniques are used to identify code reuse in malware families, which can identify the malware products produced by specific malware authors [10].

Following the identification of new malware variants, detailed analysis is required to determine the new malware capabilities. Function similarity provides the first stage of this analysis and allows the elimination of unmodified functions from the reverse engineering workload [11].

Function similarity methods are also used for the analysis of compiled programs in cases where software plagiarism or breach of licensing conditions is suspected [8].

This paper provides two studies. The first study uses a Zeus malware dataset; the second study uses a dataset from a sample of ISFB banking malware [12]. The distinct and independent codebases of the Zeus and ISFB malware families have both been publicly leaked. The Zeus dataset study highlights the improved machine learning performance. Previous research [1] was re-run using the same testing configuration as the other experiments in this paper. This gave an F1 score of 0.19 for Zeus function similarity using the CVCFS algorithm [1], FSFC gives an average F1 score of 0.44 on the Zeus dataset with the same SVM model.

The ISFB study uses the training data from the Zeus dataset to predict function similarity in a dataset taken from a pair of ISFB variants where the compilation dates differ by approximately one year. In the ISFB study, FSFC gives an average F1 score of 0.34. This study shows that the SVM model abstracts features from the Zeus dataset sufficiently to predict function pairs in an independent malware family.

This paper makes the following contributions:

- An SVM model that can abstract function similarity features from one pair of program variants to identify similar functions in an unrelated pair of program variants.
- A one-step function similarity technique that extracts features from each function under consideration and specific related functions. This contextual similarity technique strengthens the function similarity results and provides a substantial performance improvement over and above training with features taken from individual functions and provides a low false positive rate.
- Improved performance of numeric features through the use of an improved encoding of these features.
- Run time reduction, as due to the higher performance of FSFC, there is no need for an additional step to remove false positives.
- Development of a curated dataset of matched functions in two versions of ISFB malware for use in future research.

- Development of two labelled Interactive Disassembler (IDA) databases for ISFB malware versions 2.04.439, 2.16.861 (The datasets related to this research are available online at http://federation.edu.au/icsl/evolvedsimilarity).

The structure of this paper is as follows: Section 2 presents related work, Section 3 presents the research methodology, Section 4 presents the empirical evaluation of results, and Section 5 presents the conclusion.

2. Related Work

2.1. BinJuice Function Semantics

BinJuice computes abstract semantics for each basic block of all functions in a compiled program, by a process of disassembly, Control Flow Graph (CFG) generation, symbolic execution, algebraic simplification, and the computation of function semantics [13]. The aim of generating abstract semantics is to represent any two equivalent code sequences by the same semantics [14,15].

Existing tools are used to perform Disassembly and CFG extraction. Symbolic execution is used to generate basic block semantics. Symbolic execution does not involve execution on a physical processor; instead, the effects of the program instructions can be represented as a set of simultaneous equations. An algebraic simplifier provides simplification of the symbolic expressions resulting from the symbolic execution. The simplified symbolic expressions are mapped into a canonical form to ensure that sections of equivalent code having equivalent symbolic expressions are sorted in the same order. The juice operation addresses the problem of comparing sections of equivalent code containing different register allocations by renaming register names in the simplified symbolic expressions to consistent logical variables [13].

VirusBattle is built on BinJuice and is used to identify relationships between malware samples in a large database [16]. VirusBattle unpacks the submitted malware sample and creates four different representations of the unpacked code. These representations are disassembled code, generalised code, semantics, and generalised semantics (juice). VirusBattle identifies similar functions by comparing the hashes of the function's generalised semantics. This provides a significant performance gain compared with non-approximate methods, e.g., theorem solvers [15]. VirusBattle has been commercialised by Cythereal Inc and is known as Cythereal MAGIC.

Cythereal function similarity aims to identify similar functions using semantic hashes derived from symbolic execution. Cythereal similarity makes use of the abstraction of each function, and if a pair of functions are similar but have different purposes, then Cythereal will still identify these as function pairs. The design goals of FSFC are to be able to identify function similarity in evolved variants of malware families [17] and so it aims to match function pairs, including those that differ as a result of software development. The use of contextual features allows FSFC to differentiate similar functions with different purposes due to their different position in the call tree. A further point of difference is that due to FSFC's use of machine learning, there is the potential that this research could lead to a general-purpose function similarity engine that can identify similar functions in a broad range of programs.

2.2. Cross-Architecture Bug Search

DiscovRE [4] identifies vulnerable functions across a range of IoT devices. IoT devices often use open-source code and a variety of CPU architectures. Vulnerability discovery in an open-source library may impact a range of IoT devices. The identification of functions containing known vulnerabilities requires the capability to search code generated by different compilers, and a range of CPU architectures. DiscovRE uses k-Nearest Neighbors (kNN) machine learning to pre-filter potentially vulnerable functions. The Pearson product-moment correlation coefficient [18] has been used to select numeric features that are robust across multiple compilers, compiler options, and CPU architectures. The following function counts have been used as features: call instructions, logic instructions, redirection instructions, transfer instructions, local variables, basic blocks, edges, incoming calls, and instructions. The final identification of vulnerable functions is performed using maximum common subgraph isomorphism.

In some scenarios, the DiscovRE pre-screening stage is unreliable [19]. CVSkSA performs cross-architecture searching for vulnerable functions in IoT device firmware using the kNN algorithm, followed by SVM machine learning for pre-filtering [19]. Bipartite graph matching [5] is then used to identify vulnerable functions from the pre-screened matches. Although the accuracy of the SVM model for pre-filtering is good, the run time is slow, but kNN pre-filtering reduces the number of functions to be checked by the SVM model, and this reduces execution time by a factor of four with a small reduction in vulnerability identification performance. CVSkSA uses two levels of features; function level features are used in the pre-filtering while basic block level features are used for graph matching. The function level features are call count, logic instruction count, redirection instruction count, transfer instruction count, local variables size, basic block count, incoming call count, and instruction count.

The above research techniques locate software vulnerabilities in one version of code, compiled for different CPU architectures. These techniques are optimised for cross-architecture vulnerability identification within one version of a program. The research in this paper provides identification of similar functions in different versions of a program compiled for a single CPU architecture.

2.3. CVCFS

This paper builds on an earlier research work called CVCFS [1], that identifies similar function pairs in two variants of the same program. There are three areas where FSFC has improved on CVCFS. The first is an improved definition of function context, the second is a rationalisation of the features, and the third is an improved encoding of numeric features. In CVCFS, a function context was defined as the set of non Application Programming Interface (API) functions that can be reached by walking the call graph starting from the selected function. This definition could lead to a large number of functions in a function context. FSFC restricts function context and limits the call graph depth for feature extraction. This use of smaller function contexts results in substantially higher performance. CVCFS makes use of the following local features from individual functions: the set of API calls, the set of constants, stack size, function callers count, and basic block count. The following contextual features were extracted: set of API calls, set of constants, stack size, return instruction release count, and call instructions count. CVCFS uses local and contextual features that are similar but are not fully aligned. For example, CVCFS local features contain function callers count, and the contextual features contain a call count feature. FCFS removes the distinction between local and contextual features and uses six consistent features, as defined in Section 3.3. In CVCFS, numeric features from each function in the context are summed, while API call names and constants are stored in sets. During the testing of CVCFS, the performance of the numeric features was low. In FCFS, numeric features are stored as a set, and each element in the set consists of a value from the function context. In CVCFS, 20,000 training iterations were used, FCFS research found that better performance resulted from 100,000 training iterations. This paper demonstrates the substantial performance improvements obtained in FCFS by using an improved definition of function context, improved encoding of numeric features, and increased training iterations.

3. Function Similarity Using Family Context

The FSFC research is introduced in the following order, experimental design, function context definition, feature definition, feature extraction, feature ratios, ground truth, and the FSFC algorithm.

3.1. Experimental Design

The FSFC method uses an SVM model to classify similar function pairs from a pair of program variants. Both the training and classification steps use features that are calculated from the pairwise comparison of all of the functions in two program variants. These features are calculated using the feature ratios described in Section 3.5. The Function Similarity Ground Truth (FSGT) table (see Section 4.2) contains the Relative Virtual Addresses (RVAs) of each matching function pair in the training and testing datasets.

The FSGT table is used to label the training data. Due to the pairwise feature generation, many of the training features correspond to unmatched function pairs, and a smaller set of features correspond to matching function pairs. As a result, the training dataset is unbalanced, and this is discussed further in Section 4.4. The training features represent exact and approximate function matches. The SVM model training uses the feature labels to identify the features corresponding to matching function pairs. The trained SVM model contains weights and biases that can be used to classify similar functions. The SVM model classification indicates whether the features from each pairwise function pair correspond to a similar function. The FSGT table is used to identify whether the similarity classification is correct for each function pair. The function pairs classified correctly as similar functions are added to the True Positive (*TP*) count. Similarly, the correctly classified dissimilar function pairs are added to the True Negative (*TN*) count. Function pairs that are incorrectly identified as similar are added to the False Positive (*FP*) count, and function pairs that are incorrectly identified as dissimilar are added to the False Negative (*FN*) count. Precision and recall are calculated from the *TP*, *TN*, *FP*, and *FN* counts. *Precision* and *recal*l [20] are defined in Equations (1) and (2), respectively.

$$Precision = TP/TP + FP \tag{1}$$

$$Recall = TP/TP + FN \tag{2}$$

The F1 score (*F1*) [20] defined in Equation (3) is used to assess the quality of the results in terms of precision and recall.

$$F1 = 2 \times (Precision \times Recall)/(Precision + Recall) \tag{3}$$

3.2. Function Context

In FSFC, function context is defined as: Self (*S*), Child (*C*), Parent (*P*), Grandchild (*GC*), Grandparent (*GP*), Great-grandchild (*GGC*) and Great-grandparent (*GGP*). Each of these function contexts is a set of associated functions.

The Self (*S*) context of function f is the set of the non-API functions that are called from function f.

$$S(f) = \{s_0, s_1, ..., s_{\mathrm{i}}\} \tag{4}$$

The set of functions in the Child *C* context of f can be obtained by iterating through each of the functions in the Self context of f and extracting the non-API functions.

$$C(f) = \{s(f') \; \forall \; f' \; in \; S(f)\} \tag{5}$$

The functions in the Grandchild *GC* context of function f can be obtained by iterating through each of the functions in the Child context of f and extracting the non-API functions.

$$GC(f) = \{s(f') \; \forall \; f' \; in \; C(f)\} \tag{6}$$

The functions in the Great-Grandchild *GGC* context of f can be obtained by iterating through each of the functions in the Grandchild context of f and extracting the non-API functions.

$$GGC(f) = \{s(f') \; \forall \; f' \; in \; C(f)\} \tag{7}$$

The functions in the Parent *P* context of f is the set of non-API functions that call function f.

$$P(f) = \{p_0, p_1, ..., p_{\mathrm{j}}\} \tag{8}$$

The functions in the Grandparent *GP* context of *f* can be obtained by identifying the callers of each of function in the Parent context of *f*.

$$GP(f) = \{P(f') \; \forall \; f' \; in \; P(f)\} \tag{9}$$

The functions in the Great-Grandparent *GGP* context of *f* can be obtained by identifying the callers of each of function in the Grandparent context of *f*.

$$GGP(f) = \{P(f') \; \forall \; f' \; in \; GP(f)\} \tag{10}$$

3.3. Features

FSFC extracts the following features from each function in the context of function *f*:

- API calls,
- Function calls count,
- Return release count,
- Constants,
- Stack size,
- Basic block count.

API calls: The system programming interface for the Windows operating system is provided by the Windows API [21]. The API provides a dynamic function call interface for Windows system services. Windows programs that use the portable executable (PE) format. Let $AC(f, p)$ be the set of API functions called by function *f* in a program *p*.

$$AC(f, p) = \{a_0, a_1, ..., a_\mathrm{k}\} \tag{11}$$

Function calls count: Let *FC* be the count of function call instructions *c* within function *f* in program *p*.

$$FC(f, p) = |\{c_0, c_1, ..., c_\mathrm{l}\}| \tag{12}$$

Return release count: Let *rc* be the byte release count of any return instruction within function *f*, and let *m* be the count of return instructions within *f*. If *rc* is 0, then *rc* is set to 1. Let *RC*, the return release count, be the sum of the release count *rc* of all return instructions within function *f* in program *p*.

$$RC(f, p) = \sum_{i=0}^{i=m} max(rc_\mathrm{i}, 1) \tag{13}$$

Constants: The goal in extracting a set of constants is to extract invariant numerical constants from the operands of instructions in functions. Call and jump instructions were excluded because they have operands that contain program and stack addresses that are not invariant. Let $CS(f, p)$ be the set of constants that are not program or stack addresses within function *f* in program *p*.

$$CS(f, p) = \{c_0, c_1, ..., c_\mathrm{n}\} \tag{14}$$

Stack size: Let $SS(f, p)$ be the stack size of function *f* in program *p*.

$$SS(f, p) = s_0 \tag{15}$$

Basic block count: A basic block is defined as the maximal sequence of consecutive instructions that begin execution at the first instruction, and when the block is executed, all instructions in the basic block are executed sequentially without halting or branching, except for the last instruction in the block [22]. Let *BB* be the count of each basic block *b* within function *f* in program *p*.

$$BB(f,p) = |\{b_0, b_1, ..., b_q\}| \tag{16}$$

3.4. Feature Extraction

CVCFS [1] summed numeric features, e.g., stack size, function callers count, function calls count, and basic block count to an individual number. In FSFC, all features are extracted from the context of a function and are stored in sets, with one value from each function in the context. For example, a context of three functions with stack sizes of 12, 0, and 24 bytes, in the earlier research, the stack size feature would be summed to 36 bytes, but in FSFC, the stack size feature would be {12, 0, 24}. Jaccard indexes provide an efficient measure of set similarity and have been used extensively in previous similarity research [2,23]. Jaccard indexes are calculated from the contextual features for each function in the pair being compared. This calculation of Jaccard indexes for all features has simplified FSFC implementation.

3.5. Feature Ratios

Feature ratios use the Cartesian product of all functions in program $p1$ and all functions in program $p2$. It is noted that function similarity is commutative, and the same function pairs will be identified by comparing programs $p1$ and $p2$ as would be identified by comparing program $p2$ and $p1$. The following features definitions provide names for the features that are used in subsequent algorithms. Let $F(p)$ be the set of all functions in program p.

$$F(p) = \{f_0, f_1, ..., f_1\} \tag{17}$$

The set of function pairs $FP(p1, p2)$ of programs *p1* and *p2* are defined as follows:

$$FP(p1, p2) = F(p1) \times F(p2) \tag{18}$$

Each element of the Cartesian product *FP* is a function pair *fp* consisting of one function $f1$ from program $p1$ and one function $f2$ from program $p2$.

$$fp = (f1, f2) \tag{19}$$

API Ratio: Let AC_1 and AC_2 be the sets of API calls extracted from the context of each function in function pair *fp*. Let the API ratio ACR be the Jaccard index of AC_1 and AC_2.

Function Calls Ratio: Let FC_1 and FC_2 be the sets of the call instructions counts extracted from the context of each function in function pair *fp*. Let the function calls ratio FCR be the Jaccard index of FC_1 and FC_2.

Return Release Count Ratio: Let RC_1 and RC_2 be the sets of return release counts extracted from the context of each function in function pair *fp*. Let the return release count ratio RCR be the Jaccard index of RC_1 and RC_2.

Constants Ratio: Let CS_1 and CS_2 be the sets of constants extracted from each of the context of each function in the function pair *fp*. Let the constants ratio CSR be the Jaccard Index of CS_1 and CS_2.

Stack Ratio: Let SS_1 and SS_2 be the sets of stack sizes extracted from the context of each function in the function pair *fp*. Let the stack ratio SSR be the Jaccard index of SS_1 and SS_2.

Blocks Ratio: Let BB_1 and BB_2 be the sets of basic block counts extracted from the context of each of the functions in the function pair *fp*. Let the blocks ratio BBR be the Jaccard index of BB_1 and BB_2.

3.6. Ground Truth

As a supervised machine learning method, FSFC requires labelled data both for training the SVM model, and for the evaluation of the system performance. To support this, the FSGT table was constructed via manual reverse engineering of the malware samples against leaked malware source code. The FSGT table contains data defining the pairing of the functions of the programs being compared. The FSGT table contains the RVAs of the function pairs.

3.7. FSFC Algorithm

The FSFC algorithm only operates on static non-API function calls. Figure 1 illustrates the relationships between the functions being processed. The function currently being processed is referred to as the current function. The self context contains those functions called by the current function. The child functions are those functions called from the self context functions. The child context consists of the function calls made from each of the child functions. The callers of the current function are referred to as parent functions. The parent context consists of the callers of each of the parent functions. The other relationships and contexts are identified in a similar manner and are defined in Section 3.2.

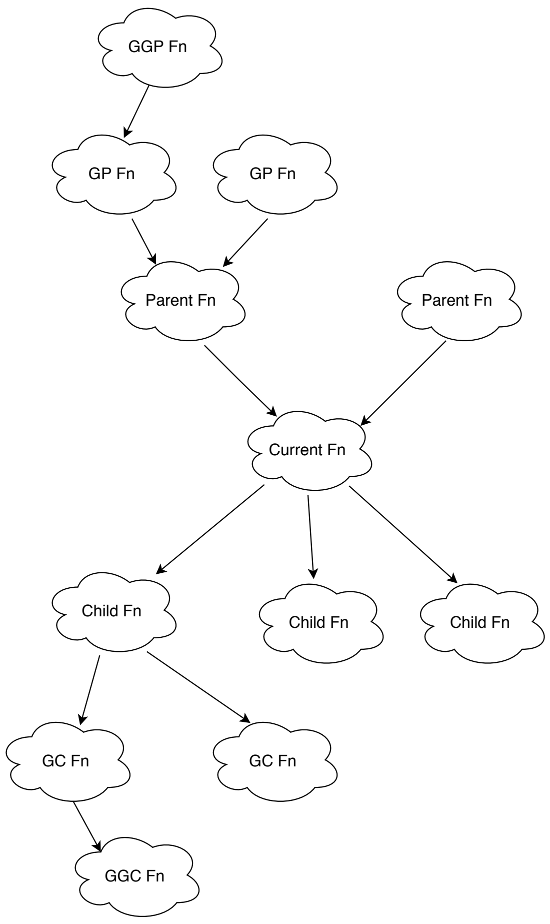


Figure 1. Function Call Relationships.

The function context extraction algorithm is shown in Figure 2. For each function *f* in the program being analysed, the self context is the set of static non-API function calls made by *f*. The child context is the set of all static non-API function calls made by each function in the self context. The grandchild and great-grandchild contexts are calculated in a similar manner. The parent contexts are built in a similar manner by walking up the call tree. The parent context $p1$ is the set of all static non-API function calls made to each function in the self context. The grandparent and great-grandparent contexts are calculated from the parent and grandparent contexts in a similar manner.

```
For each function f in program
  Build function callers map
For each function f in program
  S context c0 = static calls in f
  For each function f1 in self context c0
      C context c1 += static calls in f1
  For each f2 in child context c1
     GC context c2 += static calls in f2
  For each f3 in grandchild context c3
      GGC context c3 += static calls in f3
  From function callers map
      Build parent context P from f
  From p1
      Build grandparent context GP from P
  From p2
      Build great-grandparent context GGP from~GP
```

Figure 2. Function Context Extraction Algorithm.

The Cythereal output includes unpacked malware, disassembled code, generalized code, semantics, and generalized semantics. The FSFC algorithm reads the disassembly of the unpacked malware, extracts the functions from the disassembly, builds a map of function callers, and then iterates through each function in the program and builds the function context sets.

The algorithm for feature extraction is shown in Figure 3. This algorithm reads the Cythereal disassembly of each of the programs being compared. Next, it extracts the function contexts for each function in programs p1 and p2, and then calculates the features, AC, FC, RC, CS, SS, BB for the context of each function. For each function in program 1, and for each function in program 2, calculate the Jaccard Index of the features to produce the feature ratios ACR, FCR, RCR, CSR, SSR, BBR , then label each function combination using the FSGT table.

```
p1 = baseline version of program
p2 = updated version of~program

  for each function in p1
    extract function context for~function

  for each function in p2
    extract function context for~function

  for each function in p1
    calculate features AC1, FC1, RC1, CS1, SS1, BB1 for function~context

  for each function in p2
    calculate features AC2, FC2, RC2, CS2, SS2, BB2 for function~context

  for each f1 in p1
    for each f2 in p2
      #calculate feature ratios
      ACR = jaccard(AC1, AC2)
      FCR = jaccard(FC1, FC2)
      RCR = jaccard(RC1, RC2)
      CSR = jaccard(CS1, CS2)
      SSR = jaccard(SS1, SS2)
      BBR = jaccard(BB1, BB2)
      if f1, f2 in FSGT table
        set label = ‘‘1’’
      else
        set label = ‘‘0’’

  rva1 = rva(f1)
  rva2 = rva(f2)
  features = map(rva1, rva2, ACR, FCR, RCR, CSR, SSR, BBR)
```

Figure 3. Feature Extraction Algorithm.

4. Empirical Evaluation

The FSFC research is presented as two case studies. The first case study uses the Zeus dataset from the previous CVCFS research and provides a measure of the improvement in FSFC performance. The second case study uses the Zeus SVM model to predict similarity in a pair of ISFB malware variants. In the first experiment, the optimal number of training iterations is determined; the second experiment determines the context set that provides the best performance. The third experiment tests the performance of individual features. The fourth experiment evaluates all feature combinations to identify the highest performing feature combinations. A fifth experiment is performed using the CVCFS algorithm with the same iteration count as the FSFC research. This provides a comparison of the performance of the CVCFS and FSFC algorithms. A final experiment is performed to compare the performance of the numeric feature encoding method from CVCFS with the new numeric feature encoding in FSFC. These tests are run for the Zeus dataset and the ISFB dataset. This section concludes with a discussion of future research.

4.1. Datasets

Two malware datasets are used to evaluate FSFC; the first dataset is a Zeus dataset, and the second dataset is an ISFB dataset. Training features were extracted from Zeus samples 1 and 2. Function similarity prediction was performed on the features from Zeus samples 2 and 3 to show that the SVM model can abstract function similarity features from one pair of program variants to identify similar functions from later versions of the same malware family. Testing features were then extracted from ISFB Samples 4 and 5, and function similarity prediction was performed using the training features from the Zeus samples 1 and 2. This test was performed to show the ability of the SVM model to identify similar functions in a pair of malware variants from a different malware family. Table 1 gives the Secure Hash Algorithm 1 (SHA1) hashes identifying the malware samples used in this paper, and Table 2 shows the malware version numbers associated with these malware samples.

Cythereal provided the unpacked Zeus samples, and Malpedia provided the unpacked ISFB samples [24]. These samples were disassembled for manual analysis using IDA. The linker date in the Portable Executable (PE) header was used to indicate the time the malware samples were created. Although the linker date can be modified by the malware author, there are no inconsistencies that suggest that the linker date has been modified. The linker dates associated with the malware samples used in this paper are shown in Table 3. The linker dates provide the following development timeline: sample 2 was compiled one month after sample 1, sample 3 was compiled six months after sample 2, and sample 5 was compiled one year after sample 4.

Table 1. Malware Sample Details.

Sample SHA1 Hash	Version
8a7faa25f23a0e72a760075f08d22a91d2c85f57	Zeus 2.0.8.7
706bf4dcf01b8eceedf6d05cf5b55a27e4ff8ef0	Zeus 2.0.8.9
30c6bb2328299a252436d2a3190f06a6f04f7e3f	Zeus 2.1.0.1
ee0d9a91f52de9b74b8de36e25c2de91d008cee5	ISFB 2.04.439
3ec5a5fe4c8e6b884b2fb9f11f9995fdaa608218	ISFB 2.16.861

Table 2. Malware Sample Versions.

Sample Identifier	Version
Zeus Sample 1	Zeus 2.0.8.7
Zeus Sample 2	Zeus 2.0.8.9
Zeus Sample 3	Zeus 2.1.0.1
ISFB Sample 4	ISFB 2.04.439
ISFB Sample 5	ISFB 2.16.861

Table 3. Malware Linker Dates.

Sample Identifier	Linker Date
Zeus Sample 1	14 September 2010
Zeus Sample 2	15 October 2010
Zeus Sample 3	24 March 2011
ISFB Sample 4	17 September 2015
ISFB Sample 5	14 September 2016

4.2. Ground Truth

The FSGT table identifies the function pairs, function names, and the function RVA for each malware sample used in this research. Manual analysis of the unpacked malware samples, using the leaked Zeus [25] and ISFB [26] source code was performed to identify the function pairs for the FSGT table. Function identification was performed based on the API calls, constants, and CFG structure. Function RVAs are used to identify functions in the Cythereal disassembly and in the research code. A summary of the function name labelling is shown in Table 4. Cythereal identified 1035 and 1133 functions in the ISFB samples, while manual analysis identified 1007 and 1098 functions in the ISFB samples. This difference arises from Cythereal's identification of code fragments that perform a jump to a system call as functions. In this research, these code fragments are regarded as compiler artifacts and are excluded from the analysis. A total of 998 function pairs were identified that were common to the two ISFB samples. Of these 998 functions, 985 function pairs were present in both the Cythereal disassembly and in the manual function pair identification.

Table 4. Function Counts and Manual Match Count.

Tool	Version	Count	Version	Count
IDA	Zeus 2.0.8.7	577	Zeus 2.0.8.9	553
Cythereal	Zeus 2.0.8.7	577	Zeus 2.0.8.9	539
IDA	Zeus 2.0.8.9	553	Zeus 2.1.0.1	601
Cythereal	Zeus 2.0.8.9	539	Zeus 2.1.0.1	601
IDA	ISFB 2.04.439	1007	ISFB 2.16.861	1098
Cythereal	ISFB 2.04.439	1035	ISFB 2.16.861	1133

4.3. Features

FSFC uses the following features:

- Set of API calls
- Set of constants
- Stack size
- Call instruction count
- Return release count
- Basic block count

The above features are extracted from the context of each function. Constants are extracted from mov, push, add, cmp, and sub instructions in each function and were stored in the set of constants feature. Ad-hoc analysis showed that these instructions contained a significant proportion of invariant operands. Program and stack addresses were further filtered by excluding values greater than the program base address. As this research is based on static analysis, the non-API function call counts used in this research are a count of static function calls. Stack size is taken from the function prologue when it is present; otherwise, it is zero. The stack size *SS* is taken from the sub instruction in the function prologue shown in Figure 4. It is noted that some compilers may not use the same idiom for their function prologue.

```
push(ebp)
mov(ebp,esp)
sub(esp,SS)
```

Figure 4. Function Prologue.

4.4. SMOTE Oversampling

The function similarity training dataset in FSFC is imbalanced due to the use of the Cartesian product comparison, which results in an unstable performance of the SVM model. Assume that the two versions of the same program are being compared, and each program contains 500 functions. The maximum number of matching function pairs is 500. The number of function pairs generated by the Cartesian product is 250,000, and the minimum number of non-matching function pairs is 249,500. The use of a Cartesian product in the generation of training features inherently leads to an imbalanced dataset. The SVM model's performance was improved using the Synthetic Minority Oversampling Technique (SMOTE) [27] to rebalance the training dataset.

4.5. SVM Model Training

Zeus samples 1 and 2 are used for SVM model training, as these two Zeus samples are similar but exhibit minor differences. Function similarity ratios are calculated for the Cartesian product of all functions in the training samples. These features are labelled as matching or not matching using the FSGT table. The SVM model is used to predict function similarity between Zeus samples 2 and 3, and to predict function similarity between ISFB samples 4 and 5.

Tests are performed for the Zeus and ISFB function similarity datasets to identify the following machine learning parameters:

- Iteration count,
- Best context combination,
- Individual feature performance,
- Best feature combination.

The features in each of these tests were assigned a binary identifier, e.g., the first feature is identified as 000001, and the second feature was identified as 000010. The use of binary identifiers allowed the numbering of individual tests, and these feature identifiers are shown in Table 5.

Table 5. Numbering of Feature Combination Tests.

Feat #	Vector	Description
1	000001	API Ratio
2	000010	Calls Ratio
4	000100	Return Release Ratio
8	001000	Constants Ratio
16	010000	Stack Ratio
32	100000	Blocks Ratio

4.6. Context Sets

A function's context is the set of functions associated with a specific function. This paper tests the question of whether it is possible to improve performance by creating sets composed of the sum of different contexts. The context sets used in this research consist of the sum of the functions from the context sets shown in Table 6.

Table 6. Naming of Context Sets.

Name	Components
First Context Set	Self
Second Context Set	Self, Child, Parent
Third Context Set	Self, Child, Parent, Grandchild, Grandparent
Fourth Context Set	Self, Child, Parent, Grandchild, Grandparent, Great-Grandchild, Great-Grandparent

4.7. Statistical Significance

Due to the stochastic nature of machine learning, the function similarity prediction results vary. To address this variation, each test in this paper is repeated 20 times, unless stated otherwise, and an average F1 score is calculated. There are cases where we need to determine whether a specific test configuration provides better results than another. To answer this question, we need to determine whether the test results are normally distributed. The three sets of Zeus function similarity results from Table 7 were tested to see if they followed a normal distribution. A Shapiro-Wilk test [28] was performed, and a p-value of 0.0173 indicated that we must reject the null -hypothesis and conclude that the results were not normally distributed. As a result, a two-tailed Wilcoxon signed-rank test [29] was used to determine whether selected test results differ at a 95% level of confidence.

4.8. Zeus Dataset Tests

The functions in Zeus samples 1 and 3 exhibit more differences due to software development than the training dataset. A testing set of function similarity features were created from the Cartesian product of all functions in samples 1 and 3. The SVM model was used to predict the matching function pairs from the testing feature set. The results of this prediction were evaluated using the FSGT table. Due to the stochastic nature of machine learning, the results from individual tests vary. The F1 score is used to assess the accuracy of the results.

The function similarity tests shown in Table 7 show the effect of varying the training iteration count. The column titled "S.D" indicates whether the current row was significantly different from the preceding row. The "p-value" column gives the p-value from the two-tailed Wilcoxon signed-rank test for this comparison. These tests show that performance increased with increasing training iterations and reached a maximum at 100,000 training iterations, which gave an average F1 score of 0.44. The two-tailed Wilcoxon signed-rank test indicated that the F1 score with 100,000 training iterations was not significantly different from the test using 50,0000 iterations.

Table 7. Zeus Dataset—Training Iteration Performance.

Iterations	Average F1	S.D	p-Value
50,000	0.31	-	-
100,000	0.44	N	0.05238
150,000	0.44	N	0.72634

The function similarity tests are shown in Table 8 and test the effect of varying combinations of function context. The tests used 100,000 training iterations and different combinations of function contexts. These results show that the best results were obtained using the second context set, giving an average F1 score of 0.44. The F1 score of the second context set test was significantly different from the first context set test.

Table 8. Zeus Dataset—Training Context Combinations.

Context Set	Average F1	S.D.	p-Value
First	0.14	-	-
Second	0.44	Y	0.00008
Third	0.29	Y	0.0091
Fourth	0.26	N	0.5619

The performance of individual features was assessed by performing function similarity classification using SVM models trained for each feature. These tests used 100,000 training iterations and the second context set. The results of this evaluation are shown in Table 9.

Table 9. Zeus Individual Feature Performance.

Test	Feature	Average F1
1	API Ratio	0.23
2	Calls Ratio	0.14
4	Return Release Ratio	0.08
8	Constants Ratio	0.10
16	Stack Ratio	0.03
32	Blocks Ratio	0.21

The best performing feature combinations are shown in Table 10. The function pair prediction results in this research vary from run to run due to the stochastic nature of machine learning. Owing to the time taken to run the SVM model for all 64 feature combinations of both datasets, the assessment of all feature combinations was only run five times.

Table 10. Zeus Dataset—Highest Performing Feature Combinations.

Test	Vector	Average F1
15	001111	0.47
37	100101	0.43
41	000111	0.40
44	101001	0.43
46	101110	0.43
47	101111	0.43
51	110011	0.44
53	110101	0.44
54	110110	0.40
63	111111	0.45

4.9. ISFB Dataset Tests

The second set of tests performed in this paper determines whether ISFB function similarity can be predicted from the Zeus SVM model. The functions in ISFB samples 4 and 5 exhibit differences due to approximately one year of software development. The SVM model was used to predict the matching function pairs in the ISFB data set. The function pair matches in the FSGT table were used to evaluate the prediction results.

The function similarity tests are shown in Table 11 and test the effect of varying the training iteration count. The tests were repeated 20 times using the second context set. The F1 score of 0.34 from 100,000 training iterations was significantly different (S.D.) from the F1 score of 0.22 for 50,000 training iterations and was not significantly different from the F1 score of 0.43 for 150,000 training iterations.

Table 11. ISFB Dataset—Training Iteration Performance.

Iterations	Average F1	S.D.	*p*-Value
50,000	0.22	-	-
100,000	0.34	Y	0.0455
150,000	0.43	N	0.0703

The function similarity tests shown in Table 12 show the effect of varying combinations of function context. The tests were repeated 20 times using 100,000 training iterations and different combinations of function contexts. The second context set gives the best performance for both the Zeus and ISFB datasets.

Table 12. ISFB Dataset—Training Context Combinations.

Context Set	Average F1	S.D.	*p*-Value
First	0.11	-	-
Second	0.38	Y	0.00008
Third	0.26	Y	0.0096
Fourth	0.16	Y	0.02642

The performance of each feature was assessed by performing function similarity classification using SVM models trained for each feature. The function pair prediction was run with 100,000 iterations and the second context set. The results of this evaluation are shown in Table 13.

Table 13. ISFB Individual Feature Performance.

Test	Feature	Average F1
1	API Ratio	0.10
2	Calls Ratio	0.12
4	Return Release Ratio	0.11
8	Constants Ratio	0.14
16	Stack Ratio	0.01
32	Blocks Ratio	0.20

The feature combinations that provided the best performance are shown in Table 14. The function pair prediction was run with 100,000 iterations and the second context set. Owing to the time taken to run the SVM model for all 64 feature combinations of both datasets, the assessment of all feature combinations was only run five times.

Table 14. ISFB Dataset—Highest Performing Feature Combinations.

Test	Vector	Average F1
7	000111	0.46
44	101100	0.49
46	101110	0.45
47	101111	0.44
54	110110	0.41
58	111010	0.40
59	111011	0.41
60	111100	0.41
61	111101	0.42
62	111110	0.43
63	111111	0.45

4.10. FSFC Evaluation

The F1 score for function similarity performance for the Zeus dataset shows an average value of 0.44, and for the ISFB dataset, an average value of 0.34. For comparison, a random classifier would result in a low F1 score due to the unbalanced distribution of the classes within these datasets. If we consider Zeus dataset 2, a random classifier would classify half of the 539 matching function pairs as matching and half as non-matching, and would similarly split the 323400 non-matching function pairs between these two classes. This would give a precision score of 269.5/(269.5 + 161700) = 0.0017, and a recall of 269.5/539 = 0.5. Combining these gives an F1 score of 0.0034. For the ISFB dataset with 1035 matching function pairs, a random classifier would give a precision score of 0.0009 and a recall score of 0.5, giving an F1 score of 0.0018. Thus, the F1 scores obtained by FSFC are well above those that would be expected using random classification.

The FSFC F1 scores provide a 57–65 percent improvement over the CVCFS algorithm. Table 8 shows the Zeus experiment with the second context set, resulting in an average of 480 true positive matches and 1384 false positive matches. Table 12 shows the ISFC experiment with the second context set, resulting in an average of 943 true positive matches and 4075 false positive matches. In the manual generation of the FSGT table, the time required to identify the function pairs in these two datasets was approximately two days for the Zeus dataset and four days for the ISFB dataset. The use of this research to generate a list of candidate similar function pairs is potentially of substantial benefit to a malware analyst.

Next, we examine the execution time performance of FSFC and show that the execution time is proportional to the product of the number of functions in the two malware variants. FSFC research made use of representative historical malware samples. Therefore, the timings reported are indicative of computational costs likely to occur with further malware analysis.

The experiments in this research were conducted on a workstation using an Intel i7-3770 3.40 GHz CPU with 32 GB of RAM. The SVM machine learning software was TensorFlow v1.15. Training features were generated using the Cartesian product of all functions in both training programs. The function counts in Table 4 from the Cythereal disassembly were used to calculate the total function pairs. All features were included in the training, 100,000 training iterations, and the second context set were used to obtain the timing details shown in Table 15. The time for SMOTE oversampling was 59 s. A summary of the feature extraction and machine learning times for the FSFC experiments is shown in Table 15. The column headings provide the following information: "Op" shows whether training or classification is being performed, "Dataset" shows the dataset, "Fn Pairs" shows the function pair count, "Feat Extr (s)" shows the feature extraction time in seconds, "Extr/Pair (µs)" shows the feature extraction time per function pair in microseconds, "SVM (s)" shows the total SVM execution time in seconds, and "SVM/Pair (µs)" shows the total SVM execution time per function pair in microseconds. The FSFC Operation times in Table 15 show that the FSFC training time corresponds to 91% of the execution time for the Zeus Dataset 2 classification and 77% of the execution time for ISFB Dataset 1 classification. The training and classification use features created by the pairwise combination of functions in the two malware variants. This use of the pairwise combination of functions results in execution times that are proportional to the product of the number of functions in the two malware variants.

Table 15. Operation Times.

Op	Dataset	Fn Pairs	Feat Extr (s)	Extr/Pair (µs)	SVM (s)	SVM/Pair (µs)
Train	Zeus Dataset 1	311,003	13	42	1330	4276
Classify	Zeus Dataset 2	323,939	14	43	112	346
Classify	ISFB Dataset 1	1,172,655	37	32	357	304

4.11. Comparison With Previous Research

The Zeus and ISFB similarity experiments were re-run using the algorithm from the CVCFS research [1]. These tests used 100,000 training iterations, all features, and were repeated 20 times. The results of these tests are shown in Table 16. These test results are compared with the Zeus iterations test from Table 11 that gave an average F1 score of 0.44, and the ISFB Iterations test from Table 11 that gave an average F1 score value of 0.34. The difference between these results is statistically significant and shows that the techniques used in this paper provide a significant improvement over the CVCFS research.

Table 16. Zeus and ISFB Similarity Using Previous Research.

Dataset	Average F1	S.D.	*p*-Value
Zeus	0.19	Y	0.00038
ISFB	0.12	Y	0.00030

4.12. Numeric Feature Encoding

This paper encodes numeric features as a set of values taken from each function in the context; this allows the use of a Jaccard Index for the comparison of numeric features. For example, consider a context containing three functions with 5, 3, and 6 basic blocks. The updated numeric feature encoding encodes the blocks feature as the set 5, 3, 6, while the CVCFS method represented the blocks feature as the sum of the basic block counts, which is 14. Individual feature performance using an earlier summed numeric encoding method is shown for the Zeus dataset in Table 17, and for the ISFB dataset in Table 18. These tests used 100,000 training iterations and the second training context; 20 tests were run for each dataset. These results were compared with the individual feature performance using the new numerical feature encoding in Table 9. This comparison shows that excepting the Stack Ratio feature, the new numerical feature encoding performs significantly better than the CVCFS method. In the case of the Stack Ratio feature, the Wilcoxon test was unable to calculate an accurate *p*-value.

Table 17. Zeus Previous Numeric Feature Encoding

Feature	Average F1	S.D.	*p*-Value
Calls Ratio	0.03	Y	0.00008
Return Release Ratio	0.02	Y	0.00008
Stack Ratio	0.02	-	-
Blocks Ratio	0.02	Y	0.00008

Table 18. ISFB Previous Numeric Feature Encoding.

Feature	Average F1	S.D.	*p*-Value
Calls Ratio	0.01	Y	0.00014
Return Release Ratio	0.01	Y	0.00008
Stack Ratio	0.01	-	-
Blocks Ratio	0.01	Y	0.00008

The combined feature performance using the summed numeric encoding method for the Zeus and ISFB datasets is shown in Table 19. This test used 100,000 training iterations and the second training context; all features enabled, 20 tests were run for each dataset. The "Delta F1" column shows the difference from the corresponding FSFC F1 scores. A comparison of these results with the results using the improved numeric feature encoding in Tables 7 and 11 shows that the new numeric feature encoding method substantially improves the identification of similar function pairs.

Table 19. Previous Numeric Feature Encoding.

Malware	Average F1	Delta F1	S.D.	p-Value
ISFB	0.22	−0.11	Y	0.00906
Zeus	0.27	−0.17	Y	0.01046

4.13. Future Work

Work presented in this paper can be extended as follows:

- Investigate the limits of the generality of this research,
- Further improvement of features and feature encoding,
- Test this research on datasets exhibiting a higher degree of software evolution.

5. Conclusions

The ability to match compiled functions with similar code is important for malware triage, program patch analysis, identification of library functions containing known bugs, malware authorship analysis, identification of similar function pairs in detailed malware analysis, and for plagiarism analysis. This paper uses function similarity features from one pair of program variants (Zeus malware) to find function pairs in another unrelated program (ISFB malware). SVM models are trained on contextual features that are extracted not just from the function itself, but also, from a limited set of other functions with which it has a caller and callee relationship. These new contextual features improve the accuracy in detecting function pairs and substantially reduce the false positive rate, removing the need for an additional pass to remove false negatives. The improved numerical feature representation in FSFC results in an improvement in function similarity accuracy, and allows the use of a Jaccard Index for feature ratio comparison, and simplifies the FSFC algorithm.

A major finding of this research is that the SVM model produced by FSFC can abstract function similarity features from one pair of program variants to find function pairs in another unrelated program. This new property leads to the possibility of creating generic similar function classifiers. The Zeus training dataset is relatively small, consisting of approximately 550 function pairs. However, the SVM model trained on features from this dataset was able to predict similarity in the ISFB malware family, with an average F1 score of 0.34 for the ISFB function pair identification. The same training iteration count and context set gave the best results for both the Zeus and ISFB datasets. This result indicates that the FSFC method can abstract function similarity features from Zeus malware and successfully detect similar function pairs in a pair of ISFB malware variants. Future work will examine the ability of this approach to generalise across multiple independent programs.

Author Contributions: P.B. is the main author of this paper. P.B. contributed to the development of the ideas, design of the study, theory, analysis, and writing. P.B. designed and then performed the experiments I.G., P.V., and A.L. reviewed and evaluated the experimental design and the paper. All authors read and approved the final manuscript.

Funding: This research was performed in the Internet Commerce Security Lab (ICSL), which is a joint venture with research partners Westpac, IBM, and Federation University Australia.

Acknowledgments: The authors would like to thank Cythereal (Cythereal has licensed VirusBattle from the University of Louisiana at Lafayette) for providing access to Cythereal MAGIC [30] and to the Zeus malware dataset used in this research. Paul Black is supported by an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Black, P.; Gondal, I.; Vamplew, P.; Lakhotia, A. *Identifying Cross-Version Function Similarity Using Contextual Features*; Technical Report; Federation University, Internet Commerce Security Lab.: Ballarat, Australia, 2020. Available online: https://federation.edu.au/icsl/tech-reports/icsl_techrep_2020_01.pdf (accessed on 15 July 2020).
2. Jang, J.; Brumley, D.; Venkataraman, S. Bitshred: Feature hashing malware for scalable triage and semantic analysis. In Proceedings of the 18th ACM Conference on Computer And Communications Security, Chicago, IL, USA, 21 October 2011; pp. 309–320.
3. Flake, H. Structural comparison of executable objects. In Proceedings of the 2004 Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA 2004), Dortmund, Germany, 6–7 July 2004.
4. Eschweiler, S.; Yakdan, K.; Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In Proceedings of the 2016 Network and Distributed System Security (NDSS 2016), San Diego, CA, USA, 21–24 February 2016.
5. Feng, Q.; Zhou, R.; Xu, C.; Cheng, Y.; Testa, B.; Yin, H. Scalable graph-based bug search for firmware images. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 480–491.
6. Alrabaee, S.; Debbabi, M.; Wang, L. On the feasibility of binary authorship characterization. *Digit. Investig.* **2019**, *28*, S3–S11. [CrossRef]
7. LeDoux, C.; Lakhotia, A.; Miles, C.; Notani, V.; Pfeffer, A. Functracker: Discovering shared code to aid malware forensics. In Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats, Washington, DC, USA, 12 August 2013.
8. Luo, L.; Ming, J.; Wu, D.; Liu, P.; Zhu, S. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16-22 November 2014; pp. 389–400.
9. Osorio, F.C.C.; Qiu, H.; Arrott, A. Segmented sandboxing-A novel approach to Malware polymorphism detection. In Proceedings of the 2015 10th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico, 20–22 October 2015; pp. 59–68.
10. Hong, J.; Park, S.; Kim, S.W.; Kim, D.; Kim, W. Classifying malwares for identification of author groups. *Concurr. Comput. Pract. Exp.* **2018**, *30*, e4197. [CrossRef]
11. Jilcott, S. Scalable malware forensics using phylogenetic analysis. In Proceedings of the 2015 IEEE International Symposium on Technologies for Homeland Security (HST), Waltham, MA, USA, 14–16 April 2015; pp. 1–6.
12. Black, P.; Gondal, I.; Layton, R. A survey of similarities in banking malware behaviours. *Comput. Secur.* **2018**, *77*, 756–772. [CrossRef]
13. Lakhotia, A.; Preda, M.D.; Giacobazzi, R. Fast location of similar code fragments using semantic 'juice'. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, 26 January 2013; p. 5.
14. Ng, B.H.; Prakash, A. Expose: Discovering potential binary code re-use. In Proceedings of the 2013 IEEE 37th Annual Computer Software and Applications Conference (COMPSAC), Kyoto, Japan, 22–26 July 2013; pp. 492–501.
15. Lakhotia, A.; Black, P. Mining malware secrets. In Proceedings of the 2017 12th International Conference on Malicious and Unwanted Software (MALWARE), Fajardo, Puerto Rico, 11–14 October 2017; pp. 11–18.
16. Miles, C.; Lakhotia, A.; LeDoux, C.; Newsom, A.; Notani, V. VirusBattle: State-of-the-art malware analysis for better cyber threat intelligence. In Proceedings of the 2014 7th International Symposium on Resilient Control Systems (ISRCS), Denver, CO, USA, 19–21 August 2014; pp. 1–6.
17. Black, P.; Gondal, I.; Vamplew, P.; Lakhotia, A. Evolved Similarity Techniques in Malware Analysis. In Proceedings of the 2019 18th IEEE International Conference On Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE), Rotorua, New Zealand, 5–8 August 2019; pp. 404–410.
18. Puth, M.T.; Neuhäuser, M.; Ruxton, G.D. Effective use of Pearson's product–moment correlation coefficient. *Anim. Behav.* **2014**, *93*, 183–189. [CrossRef]

19. Zhao, D.; Lin, H.; Ran, L.; Han, M.; Tian, J.; Lu, L.; Xiong, S.; Xiang, J. CVSkSA: Cross-architecture vulnerability search in firmware based on kNN-SVM and attributed control flow graph. *Softw. Qual. J.* **2019**, *27*, 1045–1068. [CrossRef]
20. Lipton, Z.C.; Elkan, C.; Naryanaswamy, B. Optimal thresholding of classifiers to maximize F1 measure. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*; Springer: Berlin/Heidelberg, Germany, 2014; pp. 225–239.
21. Russinovich, M.E.; Solomon, D.A.; Ionescu, A. *Windows Internals*; Pearson Education: New York, NY, USA, 2012.
22. Lam, M.; Sethi, R.; Ullman, J.; Aho, A. *Compilers: Principles, Techniques, and Tools*; Pearson Education: New York, NY, USA 2006.
23. Bayer, U.; Comparetti, P.M.; Hlauschek, C.; Kruegel, C.; Kirda, E. Scalable, behavior-based malware clustering. In Proceedings of the NDSS 2009, San Diego, CA, USA, 8–11 February 2009; Internet Society: Reston, VA, USA, 2009; Volume 9, pp. 8–11.
24. Plohmann, D.; Clauss, M.; Enders, S.; Padilla, E. Malpedia: A Collaborative Effort to Inventorize the Malware Landscape. *J. Cybercrime Digit. Investig.* **2018**, *3*, 1–19
25. Zeus Author. Zeus Source Code. 2011. Available online: https://github.com/Visgean/Zeus (accessed on 15 July 2020).
26. ISFB Author. ISFB Source Code. 2010. Available online: https://github.com/t3rabyt3/Gozi (accessed on 15 July 2020).
27. Chawla, N.V.; Bowyer, K.W.; Hall, L.O.; Kegelmeyer, W.P. SMOTE: Synthetic minority over-sampling technique. *J. Artif. Intell. Res.* **2002**, *16*, 321–357. [CrossRef]
28. Shapiro, S.S.; Wilk, M.B. An analysis of variance test for normality (complete samples). *Biometrika* **1965**, *52*, 591–611. [CrossRef]
29. Wilcoxon, F.; Katti, S.; Wilcox, R.A. Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test. *Sel. Tables Math. Stat.* **1970**, *1*, 171–259.
30. Cythereal Inc. Cythereal MAGIC. Available online: http://www.cythereal.com (accessed on 15 July 2020).

Chapter 8

API Based Discrimination of Ransomware and Benign Cryptographic Programs

An understanding of malware internal operations is a prerequisite to the experimental design of malware research. Benign cryptographic programs may have some features of ransomware, and then these benign programs could be misclassified as ransomware. Previous research has proposed methods for ransomware detection using machine learning techniques. However, this research has not examined the precision of ransomware detection. While existing techniques show an overall high accuracy in detecting novel ransomware samples, previous research does not investigate the discrimination of novel ransomware from benign cryptographic programs. Machine learning based techniques would be limited in their practical benefit if they generated too many false positives, or if they deleted/quarantined critical data. This chapter implements a ransomware detection system based on an SVM model using API profile features and then examines the ability of existing API profile-based machine learning techniques to discriminate novel ransomware from benign-cryptographic programs.

The contributions of this chapter are presented as a research paper:

- P. Black, A. Sohail, I. Gondal, J. Kamruzzaman, P. Vamplew, and P. Watters, "API Based Discrimination of Ransomware and Benign Cryptographic Programs," *ICONIP 2020*, ERA A, 2020 (Accepted).

API Based Discrimination of Ransomware and Benign Cryptographic Programs

Paul Black[1]( ), Ammar Sohail[1], Iqbal Gondal[1], Joarder Kamruzzaman[1], Peter Vamplew[1], and Paul Watters[2]

[1] Internet Commerce Security Lab, Federation University, Ballarat, Australia
{p.black,iqbal.gondal,joarder.kamruzzaman,p.vamplew}@federation.edu.au, ammarsohail@gmail.com

[2] Cybersecurity and Networking Group, Latrobe University, Melbourne, Australia
p.watters@latrobe.edu.au

Abstract. Ransomware is a widespread class of malware that encrypts files in a victim's computer and extorts victims into paying a fee to regain access to their data. Previous research has proposed methods for ransomware detection using machine learning techniques. However, this research has not examined the precision of ransomware detection. While existing techniques show an overall high accuracy in detecting novel ransomware samples, previous research does not investigate the discrimination of novel ransomware from benign cryptographic programs. This is a critical, practical limitation of current research; machine learning based techniques would be limited in their practical benefit if they generated too many false positives (at best) or deleted/quarantined critical data (at worst). We examine the ability of machine learning techniques based on Application Programming Interface (API) profile features to discriminate novel ransomware from benign-cryptographic programs. This research provides a ransomware detection technique that provides improved detection accuracy and precision compared to other API profile based ransomware detection techniques while using significantly simpler features than previous dynamic ransomware detection research.

Keywords: Ransomware · Machine learning · Internet security and privacy · Dynamic analysis

1 Introduction

Ransomware seeks to encrypt user data or lock the victim's computer and then extort money to regain access. Although early ransomware programs were detected in 2006, the frequency of ransomware attacks have recently accelerated and have become a significant information security problem [1]. The UK National Health Service (NHS) suffered a ransomware attack in 2017, 80 out of 236 NHS trusts were infected by ransomware; the full cost of this attack is not known [2].

H. Yang et al. (Eds.): ICONIP 2020, LNCS 12533, pp. 1–12, 2020.
https://doi.org/10.1007/978-3-030-63833-7_15

Tangible and non-tangible losses due to ransomware extend the impact beyond reported cash losses. These costs include investigation, new anti-ransomware strategies, data recovery, forensic costs, legal costs, crisis communication, fines, revenue loss, and reputational damage [3].

Prior research [4–8] has shown that novel ransomware families may be detected through the use of machine learning techniques trained on the features of existing ransomware samples. However, this research does not consider that some common programs perform similar cryptographic operations to ransomware. While existing techniques show an overall high accuracy in detecting novel ransomware samples, to the best of our knowledge, none of the previous research investigates the discrimination of ransomware from common programs that share some of the cryptographic characteristics of ransomware. This is a critical, practical limitation of current research; machine learning based techniques would be limited in their practical benefit if they generated too many false positives (at best) or deleted/quarantined critical data (at worst).

The research in this paper provides significantly improved accuracy and precision in ransomware detection when compared with existing Application Programming Interface (API) profile based techniques [7] and does so using simplified features. The experiments in this paper were performed using dynamic analysis of ransomware and benign programs in a Cuckoo sandbox. An SVM model was trained using the API profile extracted from the Cuckoo results for each program. Feature selection was used to select the highest performing API features.

1.1 Command and Control Server Emulation

Historical ransomware samples were used in this research; a Command and Control (C2) server is used by a ransomware sample to obtain a public encryption key and to report successful infections. A common technique to disable a ransomware attack is to remove the Domain Name Server (DNS) entry for the C2 server. A consequence of this is historical ransomware samples may be executed in a virtual machine (VM), but network access to the C2 server will not be available. A C2 emulator is a program that is written to emulate the operation of the absent C2 server. This use of a C2 emulator allows a historical malware sample to exchange the required information with the emulated C2 server and to proceed to searching for and encrypting user files. Running historical malware samples with a C2 emulator provides accurate API profiles containing API calls related to initialization, network communication, file searching, and file encryption.

1.2 Contribution

The contributions of this paper are as follows:

- This research is the first to consider whether machine learning API profile based ransomware detection techniques could distinguish common cryptographic programs from ransomware. In neglecting this question, previous

research has focused on detection accuracy but has failed to consider precision as a performance metric.
- Development of techniques to improve the discrimination of ransomware from benign-cryptographic programs. Experimentation showed that API based machine learning has difficulties in discriminating ransomware from benign-cryptographic programs.
- C2 emulators for Cryptowall and CryptoLocker were developed to allow samples of these ransomware families to be run in a simulated environment and to perform the full range of operations that were possible when the malware attack was active.

The remainder of the paper is organized as follows: related work is reviewed in Sect. 2, Sect. 3 presents our research methodology. Section 4 covers feature selection and machine learning. We perform a detailed evaluation of our methodology in Sect. 5, and Sect. 6 concludes the paper.

2 Related Work

Malware analysis may be performed using either static or dynamic analysis. Both static and dynamic analysis techniques have been used to extract specifications of malware behaviour [9,10]. The methodology proposed in this paper uses dynamic analysis.

2.1 Ransomware Detection

Existing research deals with detecting ransomware using machine learning [4–8,11]. *EldeRan* [4] demonstrates the importance of feature selection to reduce the overall complexity of the problem and to improve the performance of machine learning. *EldeRan* uses features from the following classes: API calls, registry key operations, file system operations, file operations per file extension, directory operations, dropped files, and strings. The dataset used in this research consisted of 582 samples of ransomware belonging to 11 malware families and 942 benign programs. The benign programs consisted of generic utilities for Windows, drivers, browsers, file utilities, multimedia tools, developer's tools, network utilities, paint utilities, databases, emulator and virtual machine monitors, office tools. While this is a comprehensive dataset, it does not specifically target programs with cryptographic features that could be misclassified as ransomware. Experiments were performed to test the ability of *EldeRan* to detect known ransomware, and to detect novel ransomware. Testing with known ransomware provided an average accuracy of 0.963, and testing with novel ransomware samples gave a detection rate of 0.933 with 100 features and a detection rate of 0.871 with 400 features [11].

The research in [7] uses Windows API call data from the Cuckoo sandbox to generate a vector model of API calls to train an SVM machine learning model for ransomware detection. This research uses a vector representation that encoded the API call logs using a q-gram frequency and a standardized vector representation. The research uses 312 samples of benign software, further details of these programs are not provided, 276 ransomware programs targeting the Windows Operating System are used in this research. This dataset includes WannaCry, Cerber, Petya, and CryptoLocker, but further details are not provided. The accuracy of this research using the proposed vector format was 0.9352, and 0.9748 using an extension to vector encoding technique. The published results do not include true positive or false positive values. It is noted that the malware samples were not divided into malware families before splitting for training/testing, this allows the possibility that samples of malware families present in the training data were also present in the test data, raising the apparent average detection accuracy.

RansHunt is a hybrid analysis system that used static and dynamic analysis for ransomware detection. RansHunt uses the following feature classes: function length frequency, strings, API calls, registry operations, process operations, and network operations [11]. The dataset used in this research consisted of 360 samples of ransomware from 21 families, 532 different types of malware, and 460 benign software. Details of the types of benign software in the dataset were not provided. This paper uses a 10 fold cross-validation approach. Performing coss-validation selection at the ransomware family level would give a better understanding of research performance. This would avoid the possibility of having samples of the same malware families in both the train and test datasets. Feature selection was performed using Mutual Information criteria. The accuracy and precision values for static analysis were 0.935/0.951, for dynamic analysis was 0.961/0.960 and were 0.971/0.970 using the hybrid approach.

An analysis of the API calls made by malware samples from 14 malware families concluded that it may be feasible to identify ransomware behaviour on the basis of API call profile data alone [6].

GURLS [8] uses API call frequency features and machine learning based on Regularized Least Squares (RLS) for ransomware detection. The highest average binary detection rate of 0.886 was achieved using a radial basis (RBF) kernel. Multiclass classification was used to identify each ransomware family with an average accuracy of 0.867.

3 Research Methodology

The user mode programming interface for the Windows operating system is provided by the Windows API [12]. API calls in malware are readily identified by dynamic analysis techniques. This allows the creation of API profiles for detection and classification [13]. The proposed method is based on the observation that malware samples execute a unique sequence of API calls that can be used to distinguish them from other programs. Our approach uses API call profiles as features and uses feature selection to determine the most significant features.

An SVM machine learning model is used as a classifier to distinguish ransomware from benign programs and benign-cryptographic programs. API calls traces are taken from three datasets (ransomware, benign programs, and benign-cryptographic programs) using the Cuckoo sandbox. The Windows API calls, and the native functions calls are extracted from the API calls trace and are represented as a vector of API call frequency values that are labelled as ransomware/non-ransomware. Mutual Information Criteria (statistical models) is used to extract the most significant features. SVM machine learning is performed on the labelled API call frequency data. The trained model is used to predict whether a sample is ransomware or non-ransomware. The motivation for the use of SVM for learning and classifying ransomware is that, for binary classification, SVM has a high generalization rate and is designed to process large datasets with large feature spaces [7,14]. In our setting, the number of features is relatively high, so linear classifiers are a better choice [15,16].

3.1 Cross-Validation Approach

A cross-validation approach was used where the ransomware samples from one malware family and an equal number of benign programs were used for testing. The remaining ransomware samples and benign programs were used for training. This process was repeated for each ransomware family, and 10 experiments were performed for each ransomware family.

3.2 C2 Emulators

These emulators allow experimentation with historical ransomware samples by providing an emulation of the C2 server used by the malware family. Simulated DNS responses were provided by the Apate DNS simulator [17]; this permits the ransomware process to perform the necessary communications with the emulated C2 server and then continue and encrypt the user files in the test environment. This emulation exercises more of the ransomware capabilities and allows a complete API profile to be collected. C2 emulators were developed for the CryptoWall and CryptoLocker ransomware families.

4 Feature Selection and Classification

The machine learning method consists of two parts: feature selection and classification. Feature selection is performed using statistical and model-based techniques. For the statistical technique, we used Mutual Information Criteria [18] using Python's Scikit-learn library and Information Gain using the Weka machine learning tool [19]. For the model-based technique, we utilized decision trees (Random Forest) [20]. These techniques enable us to choose the most significant features API features.

4.1 Feature Engineering

The detection of ransomware activities may be performed by analysing their API calls. API call frequency profiles are employed to identify ransomware behaviour in a controlled environment. Feature selection is used to identify the most significant features, allowing the generation of simpler machine learning models, reducing the training and prediction time, and helping to counteract the problem of overfitting. These techniques are not always used in machine learning malware detection approaches [15,21]. The most significant API calls are selected based on the required level of significance using Mutual Information Criteria. After carrying out several experiments, we found that the highest accuracy could be achieved by utilizing 60% of the most significant selected features.

An API call frequency profile is required for our experiments. This API call profile can be represented by vectors, where each entry is a frequency of a given API function. Let $S = \{a_1, a_2, a_3 ..., a_n\}$ be a set of all selected features (API calls). A log of an application execution can be recorded as a sequence of API calls of length l, denoted as $A = \{a_1, a_2, ...a_l\}$ where $a_i \in S$ and $l \leq n$.

Let φ be the frequency of a Windows API function, we define a function Ψ that maps A to S and transforms each program's API calls profile to a vector of dimension $|S|$ as shown in Eq. 1.

$$\boldsymbol{v_{(A)}} = \Psi(a)_{a \in S} \tag{1}$$

where

$$\Psi(a) = \begin{cases} \varphi, & \text{Frequency of an API call if present} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

5 Experiments

In this section, we implement our SVM model and test its ability to discriminate novel ransomware from benign programs and benign-cryptographic programs. We collected 162 benign programs and 14 benign-cryptographic programs. We collected 101 ransomware samples from 15 ransomware families targeting the Windows operating system from Malpedia [22]. The ransomware families used in this research are summarized in Table 1. We collected a dataset of 162 benign programs and 14 benign-cryptographic applications; these include Winzip, SHA256, Crc32, Putty, and John the Ripper. The benign-cryptographic programs that were used in this research are summarized in Table 2.

5.1 Comparison with Existing Research

To evaluate the effectiveness of our SVM model, we selected a comparison with Takeushi's SVM based ransomware detection work [7]. This research was selected for comparison due to its relatively simple ransomware detection techniques, that are still representative of current ransomware detection research. To perform

Table 1. Summary of ransomware families

Family	Year	# Samples
TorrentLocker	2014	4
CryptoFortress	2015	2
TeslaCrypt	2015	9
Locky	2016	20
CryptXXXX	2016	6
CryptoMix	2016	4
CryptoLocker	2013	4
DirCrypt	2014	5
Petya	2016	4
Cerber	2016	10
WannaCrypto	2017	5
CryptoShield	2017	2
CryptoWall	2013	21
Cryptorium	2016	2
PadCrypt	2016	3

Table 2. Summary of Benign-cryptographic programs

Family	# Samples
Cryptographic hashing	03
Error detection	02
File compression	03
Secure data removal	02
Password cracking	02
Secure network file sharing	02

this comparison, we replicated Takeuchi's Extension To Standardized Vectors encoding technique [7]. For the remainder of this paper, we refer to this encoding technique as the *Takeuchi* technique. Although we replicated the vector encoding techniques, we continued to use our cross-validation approach. We were not able to obtain a copy of the dataset used in the *Takeuchi* research. This dataset contained 276 ransomware samples. The ransomware families of these samples are not specified in the paper, and it is assumed that the families of the individual samples are not known. The result of splitting this dataset for machine learning is that training may be performed on ransomware samples that are also present in the test dataset, and this may overstate the accuracy of the technique. The ransomware dataset used in our research contained 101 ransomware samples from

15 ransomware families. We performed cross-validation using one ransomware family at a time for testing.

5.2 Experimental Setup

A Ubuntu 16.04 LTS host operating system and a virtual machine (VM) using host-only networking were used to ensure the containment and isolation of the malware experiments. This research used a TensorFlow version 1 linear SVM model, and an Adam optimizer with a learning rate of 0.001, and 20,000 training iterations. The samples were executed in a Cuckoo sandbox [23] using a Windows XP VM. A second Windows XP VM was used to run the emulated C2 server and an emulated DNS service.

Evaluation Metrics. Each of the programs in our dataset was submitted to the Cuckoo sandbox and an API profile were extracted from the sandbox analysis report. API frequency statistics were calculated, and supervised machine learning was used to predict whether the program was ransomware. The detection was recorded as successful when a ransomware sample was identified correctly (true positive) or benign/benign-cryptographic program was detected as a "not-ransomware" (true negative). The detection fails if ransomware was identified as a "not-ransomware" (false negative) or a benign/benign-cryptographic program was identified as ransomware (false positive). We evaluated the performance using 4 metrics: accuracy, precision, recall, and F1-Score. These metrics are summarized in Table 3.

Table 3. Evaluation metrics

Metrics	Expression	Description
Accuracy	$\frac{No.\ of\ correct\ predictions}{Total\ no.\ of\ predictions}$	Correct fraction of predictions
Precision	$\frac{TP}{(TP+FP)}$	Rate of relevant results (Trues)
Recall	$\frac{TP}{(TP+FN)}$	Sensitivity for the most relevant results
F1-Score	$2 \times \frac{Recall \times Precision}{Recall + Precision}$	Estimate of entire system performance

5.3 Feature Selection

Over several experiments, we found that Mutual Information and Random Forest techniques are comparable and outperformed Information Gain for feature selection. Use of the Mutual Information Criteria from *Python's Scikit-learn* machine learning library provided an 11% increase in accuracy compared with the Information Gain algorithm from Weka. We performed several experiments to determine the most significant features and evaluated our SVM model. We observed that our machine learning model performed best when the top 60% of the features were selected after being ranked by the mutual information criteria. While this machine learning approach is able to identify the highest performing set of API features, this approach does not identify the highest performing individual API features. Table 4 summarizes the number of most significant features selected in our experiments.

Table 4. Number of most significant featuress

Experiment type	# Features selected
Ransomware/Benign Programs	118
Ransomware/Benign Cryptographic Programs	90

5.4 Ransomware Against Benign Programs

In this experiment, we test the ability of our technique to distinguish ransomware from benign programs and compare the results with the replicated *Takeuchi* vector encoding technique. Table 5 provides the accuracy, precision, recall, and F1-Score measures of both the methods. Our model substantially outperformed our replication of the *Takeuchi* technique with an improvement of 6.2% in accuracy, 6.2% improvement in precision, and an improvement of 11.1% in recall.

Table 5. Average cross-validation evaluation results

	Ransomware/Benign		Ransomware/Cryptographic	
Metric	Our method	Takeuchi method [7]	Our method	Takeuchi method [7]
Accuracy	93.3%	87.1%	67.3%	57%
Precision	96.2%	90%	67.1%	60%
Recall	90.1%	79%	71.2%	60%
F1-Score	92.2%	82.1%	67.4%	60%

5.5 Ransomware Against Benign-Cryptographic Programs

In this experiment, we test the ability of our technique to distinguish ransomware from benign-cryptographic programs and compared this to the results from the replicated *Takeuchi* vector encoding technique. The summary of these results is shown in Table 5. In this experiment, our model substantially outperformed the

Takeuchi technique with an improvement of 10.3% in accuracy, 7.1% improvement in precision, and an improvement of 11.2% in recall. Our research indicates an accuracy rate of 67.3% in distinguishing ransomware from benign cryptographic programs. Two factors that may account for this relatively low result are, firstly our dataset contained a low number (11) of benign-cryptographic programs, and secondly, our cross-validation approach of testing against program features that were excluded from training, emphasises the need for the machine learning to generalise. This gives a conservative estimate of model accuracy. We acknowledge these limitations but note that our model outperforms existing research.

6 Conclusion

In this research, we developed a technique that detects ransomware with substantially simpler features than existing research. The *Takeuchi* vector encoding technique [7] was replicated, and our model was evaluated against it. The evaluation results demonstrate that our research improves prediction accuracy and is better able to discriminate ransomware from benign-cryptographic programs.

Future research could investigate why the API profiles from some ransomware families and the use of C2 emulators resulted in lower detection rates.

Based on our research, we conclude that machine learning trained on API profile features is limited in its ability to discriminate between ransomware and benign cryptographic programs due to the significant overlap of API calls profiles.

Acknowledgement. This research was funded in part through the Internet Commerce Security Laboratory (ICSL), a joint venture between Westpac, IBM, and Federation University Australia. Paul Black is supported by an Australian Government Research Training Program (RTP) Fee-Offset Scholarship through Federation University Australia. This research was partially supported by funding from the Oceania Cyber Security Centre (OCSC).

References

1. Kharraz, A., Robertson, W., Balzarotti, D., Bilge, L., Kirda, E.: Cutting the gordian knot: a look under the hood of ransomware attacks. In: Almgren, M., Gulisano, V., Maggi, F. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2015. Lecture Notes in Computer Science, vol. 9148, pp. 3–24. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20550-2_1
2. Morse, A.: Investigation: Wannacry Cyber Attack and the NHS. National Audit Office, London **31**, 2017 (2017)
3. Layton, R., Watters, P.A.: A methodology for estimating the tangible cost of data breaches. J. Inf. Secur. Appl. **19**(6), 321–330 (2014)
4. Sgandurra, D., Muñoz-González, L., Mohsen, R., Lupu, E.C.: Automated dynamic analysis of ransomware: benefits, limitations and use for detection. arXiv preprint arXiv:1609.03020 (2016)

5. Al-rimy, B.A.S., Maarof, M.A., Shaid, S.Z.M.: A 0-day aware crypto-ransomware early behavioral detection framework. In: Saeed, F., Gazem, N., Patnaik, S., Saed Balaid, A., Mohammed, F. (eds.) Recent Trends in Information and Communication Technology. IRICT 2017. Lecture Notes on Data Engineering and Communications Technologies, vol. 5, pp. 758–766. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59427-9_78
6. Hampton, N., Baig, Z., Zeadally, S.: Ransomware behavioural analysis on windows platforms. J. Inf. Secur. Appl. **40**, 44–51 (2018)
7. Takeuchi, Y., Sakai, K., Fukumoto, S.: Detecting ransomware using support vector machines. In: Proceedings of the 47th International Conference on Parallel Processing Companion, p. 1. ACM (2018)
8. Harikrishnan, N., Soman, K.: Detecting ransomware using gurls. In: 2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC), pp. 1–6. IEEE (2018)
9. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (S&P'05), pp. 32–46. IEEE (2005)
10. Black, P., Gondal, I., Layton, R.: A survey of similarities in banking malware behaviours. Comput. Secur. **77**, 756–772 (2018)
11. Hasan, M.M., Rahman, M.M.: RansHunt: a support vector machines based ransomware analysis framework with integrated feature set. In: 2017 20th International Conference of Computer and Information Technology (ICCIT), pp. 1–7. IEEE (2017)
12. Russinovich, M.E., Solomon, D.A., Ionescu, A.: Windows internals. Pearson Education (2012)
13. Qiao, Y., Yang, Y., He, J., Tang, C., Liu, Z.: CBM: free, automatic malware analysis framework using api call sequences. In: Sun, F., Li, T., Li, H. (eds.) Knowledge Engineering and Management. Advances in Intelligent Systems and Computing, vol. 214, pp. 225–236. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-37832-4_21
14. Islam, R., Tian, R., Batten, L.M., Versteeg, S.: Classification of malware based on integrated static and dynamic features. J. Netw. Comput. Appl. **36**(2), 646–656 (2013)
15. Kolter, J.Z., Maloof, M.A.: Learning to detect and classify malicious executables in the wild. J. Mach. Learn. Res. **7**, 2721–2744 (2006)
16. Shafiq, mz., Tabish, S.M., Mirza, F., Farooq, M.: PE-miner: mining structural information to detect malicious executables in realtime. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) Recent Advances in Intrusion Detection. RAID 2009. Lecture Notes in Computer Science, vol. 5758, pp. 121–141. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04342-0_7
17. Apatedns:control your responses. https://www.fireeye.com/services/freeware/apatedns.html
18. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley, New York (2012)
19. Weka 3 data mining with open source machine learning software in java. https://www.cs.waikato.ac.nz/ml/weka/
20. Feature selection using random forest. https://towardsdatascience.com/feature-selection-using-random-forest-26d7b747597f
21. Rieck, K., Trinius, P., Willems, C., Holz, T.: Automatic analysis of malware behavior using machine learning. J. Comput. Secur. **19**(4), 639–668 (2011)

22. Plohmann, D., Clauss, M., Enders, S., Padilla, E.: Malpedia: a collaborative effort to inventorize the malware landscape. J. Cybercrime Digit. Invest. **3**(1) (2018). https://journal.cecyf.fr/ojs/index.php/cybin/article/view/17
23. Cuckoo foundation: Cuckoo sandbox - automated malware analysis. https://cuckoosandbox.org/

Chapter 9

Conclusions

9.1 Introduction

This thesis focuses on automated techniques for the analysis of banking malware and their behaviours, an emulated environment for dynamic malware analysis, techniques to identify similar malware functions, and a technique for the detection of ransomware. The first step in this analysis is a literature review focusing on banking malware. The specifics of malware families and their operation are largely found in commercial literature. A problem with a reliance on commercial literature lies in the commercial motivation, as data may be lost due to the abandoning of aging webpages when they no longer meet commercial goals. Further, quality is variable, and reports may not be complete. A review of existing literature found that malware authors reuse code and provide well-defined products, resulting in the development of distinct and persistent malware families. The research in this thesis identifies the major banking malware families, their variants, source code leakages, interrelationships, and provides related static analysis techniques

An emulated command and control infrastructure is needed for the dynamic analysis of malware to allow analysis of the malware capabilities. Large datasets of historical malware samples are available for countermeasures research. However, due to the age of these samples and previous anti-virus activities, their original malware infrastructure is no longer available. Dynamic analysis of historical malware samples results in the execution of the initialization functions. However, due to the absence of the original malware infrastructure, historical malware samples exit or repeatedly attempt to connect to network resources that no longer exist. The execution of historical malware samples can result in features that differ from those extracted in the wild, thus invalidating the results of any machine learning based on these features. A solution to these problems is the creation of C2 server emulators that support the execution of historical malware samples' full capabilities in a safe, isolated environment.

Existing program similarity techniques are used to filter previously analysed malware samples. Function similarity techniques identify a finer grained similarity at the level of compiled functions, and are widely used in security analysis to identify program updates in security patches, to identify related and updated functions in new malware variants, to exclude unchanged functions, and to identify code reuse across previously unrelated malware families. This thesis develops a machine learning based function similarity technique in three stages. An ad-hoc function similarity technique is developed in EST. In CVCFS, the ad-hoc similarity methods are replaced with an SVM model, improving generality. FSFC uses feature engineering to substantially improve performance. A major focus of this thesis has been the development of machine learning based function similarity methods with novel feature encoding techniques that increase feature strength and can identify similar functions in programs unrelated to the training set.

9.2 Static Analysis and Banking Malware Families

Chapter 3 provides the first contribution of this thesis. This contribution documents six banking malware families, their timeline, behaviours, variants, relationships, implementation, and source code leaks. Malware research has focused on techniques for malware detection, classification, clustering, and malware similarity. Prior research has not focused on specific malware families and the evolution of these malware families. Chapter 3 shows that banking malware families have become persistent, are evolving, and that knowledge of these malware families is relevant to both academic, and industry-based malware researchers. This chapter identifies the information longevity problems associated with reliance on commercial malware analysis literature and provides an analysis of selected malware families. Chapter 3 identifies 14 critical malware behaviours, specifies how these behaviours are implemented in the selected banking malware families, and then presents implementation of a static analysis technique that identifies API misuse.

9.3 Emulated Execution Environment

Chapter 4 provides the second contribution of this thesis. This contribution identifies and addresses a common problem encountered when using dynamic analysis to extract features from historic malware families. This problem arises from the removal of the malware C2 servers by anti-virus operations. A failure to consider the absence of the historical malware samples' C2 servers limits the collection of features to malware initialization; this results in the collection of features that differ from those extracted in-the-wild. Chapter 4 provides a study on the limitations of semi-automated methods for the creation of C2 servers and offers detailed examples of the construction of C2 emulators for three malware families by dynamic analysis.

9.4 Ad-hoc Function Similarity

Chapter 5 develops a technique for strengthening function similarity features when the variants being compared may differ due to software development. This technique provides the origin of the contextual features, which are developed in chapters 6 and 7. Chapter 5 provides an evolved similarity technique (EST) to identify function similarity in two variants of a malware family. A challenge in developing EST lies in the fact that one of the functions being compared may have been modified in an arbitrary manner. The solution to this problem is to extract features from the set of all non-API function calls made by the function under consideration and its associated call graph.

9.5 Similarity Using Machine Learning

Chapter 6 provides research called Cross-Version Function Similarity Using Contextual Features (CFCFS). The contribution of CVCFS is a new type of feature encoding called contextual features. The context of a specific function *f* is function *f*, plus the set of all non-API functions that can be reached by walking the call graph starting from function *f*. Contextual features are built from each of the functions in the function context. CVCFS results show that contextual feature encoding provides higher performance than features extracted from individual functions. Additional contributions are the creation of a set of labelled IDA databases for Zeus malware versions 2.0.8.7, 2.0.8.9, and 2.1.0.1 and the development of a ground truth table of matching function pairs, which will be of value for facilitating future research on malware function similarity.

9.6 Abstraction of Function Similarity

Chapter 7 provides research called Function Similarity Using Family Context (FSFC). The SVM model produced by FSFC by training on function pairs of one program (Zeus) generalizes to find similar functions in other, unrelated programs (ISFB). This new property, if validated by a larger study, leads to the possibility of creating generic similar function classifiers that can be packaged and distributed in reverse engineering tools such as IDA Pro and Ghidra.

9.7 Contextual Function Similarity

FSFC provides a formulation of function context based on an identified depth of the call graph. Specific sets of function context provide higher performance than individual function contexts and previous work. These new contextual features strengthen the function similarity results and substantially improve the performance over and above training with features taken from individual functions.

9.8 Ransomware and Benign Cryptographic Software

Chapter 8 provides research using dynamic analysis that extracts API call features for the identification of ransomware. This research focuses on the ability of ransomware detection programs to discriminate between ransomware and benign cryptographic software. Prior research using API features for ransomware detection was replicated for comparison purposes. The research presented in chapter 8 substantially outperformed previous API feature based ransomware detection research with improved accuracy, precision, and recall. Our research demonstrated that further improvement is needed in API profile-based techniques for ransomware detection to avoid misclassification of benign cryptographic software as ransomware.

9.9 Future Work:

The research in this thesis demonstrates that an SVM model trained on function pair features from a Zeus dataset was able to predict function pairs in an unrelated ISFB malware dataset. This suggests the possibility of building a generic machine learning function similarity classifier. The first step towards this classifier would be to validate the research findings in this thesis with a larger study. This proposed study does not need to be performed on malware datasets as it will be investigating similarity in general compiled programs. This research could involve the compilation of open source code with a variety of compilers. Symbolic debugging symbols from the compiled code could be used in the automatic creation of the function similarity ground truth table. This study would examine the function similarity process, test training and prediction with software compiled by different compilers, and different CPU architectures.

While previous studies have used features that capture information about each function, the refinement of the contextual features in this thesis improved the performance of function similarity identification by adding information from related functions and substantially decreased the false positive rate. The further development of contextual features to capture additional call graph information will lead to further improvement of this technique.

This thesis argues that the dynamic analysis of historical malware samples without C2 server emulation may only capture features from the malware initialisation and does not support execution of the full malware capabilities. While this may fulfill the needs for machine learning based malware detection, there are use cases where an emulated C2 server is needed to provide control of the full capabilities of the malware sample. Semi-automated methods for the creation of C2 server emulators is a new research area that merits attention.

9.10 Research Summary

The research in this thesis starts with an analysis of the nature of banking malware, and examines the major malware families, behaviours, variants, and interrelationships. In doing this, this thesis takes a broader view than other research that treats malware in a more abstract sense as only a source of features. The disadvantage of approaching malware analysis without domain knowledge is that important questions may not be considered. This thesis provides two examples where previous research has failed to consider analysis requirements adequately. The first example uses dynamic analysis to extract features from historical malware samples. A failure to recognise the malware's interaction with previously removed malware infrastructure will result in the malware executing only its initialization functions and not exhibiting its full capabilities. A second example occurs where a machine learning technique is used to identify ransomware without considering whether common benign programs may exhibit characteristics that are similar to ransomware programs and as a result may be misclassified.

The work in this thesis shows that a failure to consider malware analysis domain knowledge can lead to weaknesses in experimental design that may result in misleading conclusions. It is desired that the outcomes of the work in this thesis will lead to a closer collaboration between academic researchers and industry-based malware analysts.

Bibliography

[1] A. O. Almashhadani, M. Kaiiali, S. Sezer, and P. O'Kane, "A multi-classifier network-based crypto ransomware detection system: A case study of locky ransomware," *IEEE Access*, vol. 7, pp. 47 053–47 067, 2019.

[2] G. Wangen, "The role of malware in reported cyber espionage: a review of the impact and mechanism," *Information*, vol. 6, no. 2, pp. 183–211, 2015.

[3] P. Samuelson and S. Scotchmer, "The law and economics of reverse engineering," *Yale LJ*, vol. 111, p. 1575, 2001.

[4] P. Black, I. Gondal, and R. Layton, "A survey of similarities in banking malware behaviours," *Computers & Security*, vol. 77, pp. 756–772, 2018.

[5] J. Suaboot, Z. Tari, A. Mahmood, A. Y. Zomaya, and W. Li, "Sub-curve hmm: A malware detection approach based on partial analysis of api call sequences," *Computers & Security*, vol. 92, p. 101773, 2020.

[6] L. Borzacchiello, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Reconstructing c2 servers for remote access trojans with symbolic execution," in *International Symposium on Cyber Security Cryptography and Machine Learning.* Springer, 2019, pp. 121–140.

[7] M. J. Haber and D. Rolls, "Indicators of compromise," in *Identity Attack Vectors.* Springer, 2020, pp. 103–105.

[8] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: feature hashing malware for scalable triage and semantic analysis," in *Proceedings of*

the 18th ACM conference on Computer and communications security. ACM, 2011, pp. 309–320.

[9] H. Flake, "Structural comparison of executable objects," 2004.

[10] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code." in *NDSS*, 2016.

[11] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 480–491.

[12] C. LeDoux, A. Lakhotia, C. Miles, V. Notani, and A. Pfeffer, "Functracker: Discovering shared code to aid malware forensics," in *Presented as part of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2013.

[13] S. Alrabaee, M. Debbabi, and L. Wang, "On the feasibility of binary authorship characterization," *Digital Investigation*, vol. 28, pp. S3–S11, 2019.

[14] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu, "Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 389–400.

[15] F. C. C. Osorio, H. Qiu, and A. Arrott, "Segmented sandboxing-a novel approach to malware polymorphism detection," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2015, pp. 59–68.

[16] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[17] S. Jilcott, "Scalable malware forensics using phylogenetic analysis," in *2015 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, 2015, pp. 1–6.

[18] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *SSTIC*, vol. 5, no. 1, p. 3, 2005.

[19] D. Zhao, H. Lin, L. Ran, M. Han, J. Tian, L. Lu, S. Xiong, and J. Xiang, "Cvsksa: cross-architecture vulnerability search in firmware based on knn-svm and attributed control flow graph," *Software Quality Journal*, vol. 27, no. 3, pp. 1045–1068, 2019.

[20] D. Sgandurra, L. Muñoz-González, R. Mohsen, and E. C. Lupu, "Automated dynamic analysis of ransomware: Benefits, limitations and use for detection," *arXiv preprint arXiv:1609.03020*, 2016.

[21] B. A. S. Al-rimy, M. A. Maarof, and S. Z. M. Shaid, "A 0-day aware crypto-ransomware early behavioral detection framework," in *International Conference of Reliable Information and Communication Technology*. Springer, 2017, pp. 758–766.

[22] N. Hampton, Z. Baig, and S. Zeadally, "Ransomware behavioural analysis on windows platforms," *Journal of information security and applications*, vol. 40, pp. 44–51, 2018.

[23] Y. Takeuchi, K. Sakai, and S. Fukumoto, "Detecting ransomware using support vector machines," in *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM, 2018, p. 1.

[24] N. Harikrishnan and K. Soman, "Detecting ransomware using gurls," in *2018 Second International Conference on Advances in Electronics, Computers and Communications (ICAECC)*. IEEE, 2018, pp. 1–6.

[25] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, "A layered architecture for detecting malicious behaviors," in *Recent Advances in Intrusion Detection*. Springer, 2008, pp. 78–97.

[26] I. Kirillov, D. Beck, and P. Chase, "The maec language, overview," 2014. [Online]. Available: http://maecproject.github.io/documentation

[27] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security.* Springer, 2008, pp. 1–25.

[28] P. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," *Procedia Computer Science*, vol. 46, pp. 804–811, 2015.

[29] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, "Dynamic analysis of malicious code," *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, 2006.

[30] T. Wüchner, M. Ochoa, M. Golagha, G. Srivastava, T. Schreck, and A. Pretschner, "Malflow: Identification of c&c servers through host-based data flow profiling," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 2087–2094.

[31] M. Christodorescu, S. Jha, and C. Kruegel, "Mining specifications of malicious behavior," in *Proceedings of the 1st India software engineering conference.* ACM, 2008, pp. 5–14.

[32] J. Mankin, "Classification of malware persistence mechanisms using low-artifact disk instrumentation," Ph.D. dissertation, Northeastern University Boston, 2013.

[33] P. K. Singh, "A physiological decomposition of virus and worm programs," Master's thesis, University of Louisiana at Lafayette, 2002.

[34] A. R. A. Grégio, V. M. Afonso, D. S. Fernandes Filho, P. L. de Geus, and M. Jino, "Toward a taxonomy of malware behaviors," *The Computer Journal*, vol. 58, no. 10, pp. 2758–2777, 2015.

[35] C. Y. Cho, D. Babi ć, E. C. R. Shin, and D. Song, "Inference and analysis of formal models of botnet command and control protocols," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 426–439.

[36] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson, "Gq: Practical containment for measuring modern malware systems," in *Proceedings*

of the 2011 ACM SIGCOMM conference on Internet measurement conference, 2011, pp. 397–412.

[37] Inetsim.org, “Inetsim: Internet services simulation suite,” 2020. [Online]. Available: https://www.inetsim.org

[38] P. Barford and M. Blodgett, “Toward botnet mesocosms.” *HotBots*, vol. 7, pp. 6–6, 2007.

[39] J. Calvet, C. R. Davis, J. M. Fernandez, J.-Y. Marion, P.-L. St-Onge, W. Guizani, P.-M. Bureau, and A. Somayaji, “The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet,” in *Proceedings of the 26th Annual Computer Security Applications Conference*, 2010, pp. 141–150.

[40] F. Weyne, “Imaginary c2,” 2020. [Online]. Available: https://github.com/felixweyne/imaginaryC2

[41] A. Alwabel, H. Shi, G. Bartlett, and J. Mirkovic, “Safe and automated live malware experimentation on public testbeds,” in *7th Workshop on Cyber Security Experimentation and Test ({CSET} 14)*, 2014.

[42] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant, “Decaf: A platform-neutral whole-system dynamic binary analysis platform,” *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 164–184, 2016.

[43] J. C. King, “Symbolic execution and program testing,” *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[44] V. Ganesh and D. L. Dill, “A decision procedure for bit-vectors and arrays,” in *International Conference on Computer Aided Verification.* Springer, 2007, pp. 519–531.

[45] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, “Sok:(state of) the art of war: Offensive techniques in binary analysis,” in *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 2016, pp. 138–157.

[46] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting malware analysis with symbolic execution: A case study," in *International Conference on Cyber Security Cryptography and Machine Learning.* Springer, 2017, pp. 171–188.

[47] M. Bugalho and A. L. Oliveira, "Inference of regular languages using state merging algorithms with search," *Pattern Recognition*, vol. 38, no. 9, pp. 1457–1467, 2005.

[48] G. McGraw, "Silver bullet talks with halvar flake," *IEEE Security & Privacy*, vol. 9, no. 6, pp. 5–8, 2011.

[49] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop.* ACM, 2013, p. 5.

[50] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[51] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop.* ACM, 2013, p. 4.

[52] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *USENIX Security Symposium*, 2014.

[53] D. Gao, M. K. Reiter, and D. Song, "Binhunt: Automatically finding semantic differences in binary programs," in *Information and Communications Security.* Springer, 2008, pp. 238–255.

[54] B. H. Ng and A. Prakash, "Expose: Discovering potential binary code reuse," in *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual.* IEEE, 2013, pp. 492–501.

[55] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in

Machine Learning and Applications (ICMLA), 2012 11th International Conference on, vol. 1. IEEE, 2012, pp. 386–391.

[56] C. Miles, A. Lakhotia, C. LeDoux, A. Newsom, and V. Notani, "Virusbattle: State-of-the-art malware analysis for better cyber threat intelligence," in *Resilient Control Systems (ISRCS), 2014 7th International Symposium on.* IEEE, 2014, pp. 1–6.

[57] A. Lakhotia and P. Black, "Mining malware secrets," in *Malicious and Unwanted Software (MALWARE), 2017 12th International Conference on.* IEEE, 2017, pp. 11–18.

[58] M.-T. Puth, M. Neuhäuser, and G. D. Ruxton, "Effective use of pearson's product–moment correlation coefficient," *Animal behaviour*, vol. 93, pp. 183–189, 2014.

[59] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2017, pp. 363–376.

[60] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2019, pp. 309–329.

[61] M. M. Hasan and M. M. Rahman, "Ranshunt: A support vector machines based ransomware analysis framework with integrated feature set," in *2017 20th International Conference of Computer and Information Technology (ICCIT).* IEEE, 2017, pp. 1–7.

[62] N. Scaife, H. Carter, P. Traynor, and K. R. Butler, "Cryptolock (and drop it): stopping ransomware attacks on user data," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS).* IEEE, 2016, pp. 303–312.

[63] A. Azab, M. Alazab, and M. Aiash, "Machine learning based botnet identification traffic," in *2016 IEEE Trustcom/BigDataSE/ISPA.* IEEE, 2016, pp. 1788–1794.

[64] A. Azab, R. Layton, M. Alazab, and J. Oliver, “Mining malware to detect variants,” in *2014 Fifth Cybercrime and Trustworthy Computing Conference*. IEEE, 2014, pp. 44–53.

[65] J. Hakkarainen, “Malware analysis environment for windows targeted malware,” 2015.

[66] J. I. Forrester, “An exploration into the use of webinjects by financial malware,” Ph.D. dissertation, Rhodes University, 2014.

[67] A. Continella, M. Carminati, M. Polino, A. Lanzi, S. Zanero, and F. Maggi, “Prometheus: Analyzing webinject-based information stealers,” *Journal of Computer Security*, vol. 25, no. 2, pp. 117–137, 2017.

[68] F. BOSATELLI, “Zarathustra: detecting banking trojans via automatic, platform independent webinjects extraction,” 2013.

[69] M. Moniruzzaman, A. Bagirov, I. Gondal, and S. Brown, “A server side solution for detecting webinject: A machine learning approach,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2018, pp. 162–167.

[70] Zeus Author, “Zeus source code,” 2011. [Online]. Available: https://github.com/Visgean/Zeus

[71] J. Wyke, “What is zeus?” Sophos Labs, Tech. Rep., 2011. [Online]. Available: https://www.sophos.com/en-us/threat-center/technical-papers/what-is-zeus.aspx

[72] Dell Secureworks Counter Threat Unit, “Cryptowall ransomware threat analysis,” 2014. [Online]. Available: https://www.secureworks.com/research/cryptowall-ransomware

[73] D. Plohmann, M. Clauss, S. Enders, and E. Padilla, “Malpedia: A Collaborative Effort to Inventorize the Malware Landscape,” *The Journal on Cybercrime & Digital Investigations*, vol. 3, no. 1, 2018. [Online]. Available: https://journal.cecyf.fr/ojs/index.php/cybin/article/view/17

[74] K. Jarvis, "Cryptolocker ransomware," 2013. [Online]. Available: https://www.secureworks.com/research/cryptolocker-ransomware

[75] Panda Security, "Cryptolocker: What is and how to avoid it," 2015. [Online]. Available: https://www.pandasecurity.com/mediacenter/malware/cryptolocker

[76] A. Allievi, H. Unterbrink, and W. Mercer, "Cryptowall 4 - the evolution continues," 2015. [Online]. Available: https://blog.talosintelligence.com/2015/12/cryptowall-4.html

[77] Sophos, "The current state of ransomware: Cryptowall," 2015. [Online]. Available: https://news.sophos.com/en-us/2015/12/17/the-current-state-of-ransomware-cryptowall

[78] K. Cabaj and W. Mazurczyk, "Using software-defined networking for ransomware mitigation: the case of cryptowall," *Ieee Network*, vol. 30, no. 6, pp. 14–20, 2016.

[79] A. Walenstein and A. Lakhotia, "A transformation-based model of malware derivation," in *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 2012, pp. 17–25.

[80] I. U. Haq, S. Chica, J. Caballero, and S. Jha, "Malware lineage in the wild," *arXiv preprint arXiv:1710.05202*, 2017.

[81] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[82] T. Kim, Y. R. Lee, B. Kang, and E. G. Im, "Binary executable file similarity calculation using function matching," *The Journal of Supercomputing*, vol. 75, no. 2, pp. 607–622, 2019.

[83] P. Black, I. Gondal, P. Vamplew, and A. Lakhotia, "Evolved similarity techniques in malware analysis," in *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 2019, pp. 404–410.

[84] C. Cortes and V. Vapnik, “Support-vector networks,” *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[85] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows internals.* Pearson Education, 2012.

[86] M. Lam, R. Sethi, J. Ullman, and A. Aho, *Compilers: Principles, techniques, and tools.* Pearson Education, 2006.

[87] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[88] Cythereal Inc, “Cythereal magic,” 2018. [Online]. Available: https://www.cythereal.com

[89] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “Smote: synthetic minority over-sampling technique,” *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[90] C. J. Van Rijsbergen, *Information retrieval.* Citeseer, 1979.

[91] F. Guo, P. Ferrie, and T.-C. Chiueh, “A study of the packer problem and its solutions,” in *International Workshop on Recent Advances in Intrusion Detection.* Springer, 2008, pp. 98–115.

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

[93] T. Jenke, D. Plohmann, and E. Padilla, “Roamer: Robust automated malware unpacker,” 2019.

[94] Infosec Institute, “Yara: Simple and effective way of dissecting malware,” 2015. [Online]. Available: http://resources.infosecinstitute.com/yara-simple-effective-way-dissecting-malware/#gref

[95] Yara, “Yara: The pattern matching swiss knife for malware researchers,” 2017. [Online]. Available: https://virustotal.github.io/yara

[96] A. Zhdanov, "Generation of static yara-signatures using genetic algorithm," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019, pp. 220–228.

[97] Yara Rules, "The yara rules project," 2017. [Online]. Available: https://github.com/Yara-Rules/rules

[98] A. Kleymenov and A. Thabet, *Mastering Malware Analysis: The complete malware analyst's guide to combating malicious software, APT, cybercrime, and IoT attacks*. Packt Publishing Ltd, 2019.

[99] E. Eilam, *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, 2011.

[100] Volatility Foundation, "Volatility foundation," 2017. [Online]. Available: http://www.volatilityfoundation.org

[101] A. Margosis and M. E. Russinovich, *Windows Sysinternals administrator's reference*. Pearson Education, 2011.

[102] C. Sanders, *Practical packet analysis: Using Wireshark to solve real-world network problems*. No Starch Press, 2017.

[103] X64DBG Authors, "x64dbg - an open-source x64/x32 debugger for windows," 2020. [Online]. Available: https://x64dbg.com/#start

[104] C. Eagle, *The IDA Pro Book: the unofficial guide to the worldś most popular disassembler*. No Starch Press, 2008.

[105] National Security Agency, "Ghidra," 2020. [Online]. Available: https://www.nsa.gov/resources/everyone/ghidra

Appendix A

Appendix 1

A.1 An Introduction To Malware Reverse Engineering

Performing academic malware analysis without knowledge of malware internals and the basics of reverse engineering may lead to results that are not optimal. A situation considered in this thesis involved a case where machine learning is used for malware detection. In this situation, a sandbox is used for dynamic malware analysis of historic malware samples. There are two potential problems that may arise in this scenario. The first problem is that the malware C2 servers may have been removed, the second problem is that due to the malware samples being packed (encrypted), the same malware families may be present in the training and test data, and this can lead to inflated detection accuracy. A solution to these problems in experimental design may be found in providing researchers with a better understanding of the nature of malware and basic reverse engineering techniques.

A.1.1 Packing

Packing is a process where an executable program is encrypted. Packers may be used to protect intellectual property in commercial software, packers are widely used to prevent anti-virus programs from readily identifying malware. Packers are defined as *Programs that transform an input binary's appearance without affecting its execution semantics* [91].

Packing a program hinders static analysis by encrypting,or compressing the original program, thereby hiding the machine code and internal data of the original program. The packing operation appends an unpacking stub to the start of the packed program to allow the original program to be reconstructed and executed. Different packers may employ various strategies. A common strategy is to unpack the whole program to memory, while another unpacking strategy is to unpack instructions just before execution. When a packed program is executed, the unpacking stub may iterate through the packed data, decoding it to memory, rebuilding the import address table (IAT) of the original executable, and finally executing the malware from the Original Entry Point (OEP) [92].

A high percentage of malware samples are packed [91]. A program's entropy provides an indication of whether a program is packed. Common methods to unpack a malware sample are to manually unpack the malware using a debugger, to run the program in a sandbox and dump the program memory, or to use an automatic unpacker. Large numbers of packed malware samples are identified every day. Many of these malware samples are duplicates that appear unique due to encryption. When these malware samples are unpacked, the number of unique malware samples is reduced by orders of magnitude [15].

Datasets of historical malware samples generally consist of packed malware samples. The use of packed malware datasets has two problems. The malware family may not be known, and packed malware datasets frequently contain duplicates. In machine learning experiments, malware packing may result in the same malware variants being present in both the training data and the test data. A result of this unintentional duplication is to inflate the accuracy of test results. A more robust methodology may be to unpack the malware samples in the dataset and ensure that variants present in the training data are not also present in the test data.

Automatic unpackers operate by identifying the memory allocation that the unpacked malware was written to. While several automatic malware unpackers have been created, they are generally not publicly accessible. However, the RoAMer unpacker is available on Github [93]. Malpedia provides a curated collection of a large number of packed, and unpacked samples of malware families, their variants, and yara detection rules. [73].

```
rule Zeus_Ice_IX
{
meta:
author = "p.black@federation.edu.au"
version = "0.01"
description = "This rule detects Zeus Ice IX malware, memdumps only, Core::InitLoadModules"
strings:
    $mod_rc4 = {80 [2] 03 0F B6 [2] 8A 14 06 8D 4A 07 00 4D FE 0F B6 [2] 8A 1C 01}
condition:
    $mod_rc4
}
```

Figure A.1: Example Yara Rule

A.1.2 Malware Identification

Malware family identification is commonly performed using yara rules; these rules are regular expressions that operate on unpacked malware or memory dumps [94, 95]. Yara rules can be created manually or by using yara rule generators [96]. Yara rules are available on Malpedia [73] or in yara rule exchange forums [97]. A yara rule for detecting ICE IX malware is shown in Figure A.1.

A.1.3 Program Structure

Reverse engineering is enabled by understanding and making use of the binary structures created by the operating system and compiler authors. In the Windows operating system, the program structure is provided by the Portable Executable (PE) format [98]. The Portable Executable (PE) format is the executable file format for the Windows Operating System. The PE format supports both 32 bit and 64 bit programs. The PE format defines a header called the PE Header [99]. A program using the PE format contains two headers followed by a series of program sections. The first header is the legacy DOS header, identified by the ”MZ” signature at the start of the header, this is followed by the PE header identified by the ”PE” signature. The sections table follows the PE Header. The values in the PE header are used by the Windows loader to load the program into memory on program execution. Simple, practical uses of the PE header include identifying programs in memory using the DOS and Portable Executable signatures, location of the program entry point, and identification of compilation date. A comprehensive introduction to Windows program structures is provided in [98].

A.1.4 Memory Dump Analysis

Malware analysis is commonly performed within a VM. Benefits of using a VM include simplified physical memory dumps, and fast restoration back to an uninfected state. The Volatility Framework [100] is an open-source tool that supports the analysis of memory dumps from Windows, Mac, and Linux operating systems. The Volatility Framework uses operating system data structures to identify processes, network connection, open files, open registry keys, and to extract process memory. There are two distinct phases of malware analysis. The initial phase is performed before a yara rule is available, the second phase is performed once a yara rule has been created. In malware analysis, when a yara rule for the malware is not available, analysis is focused on the identification of the process that the malware has been injected into and on locating an unpacked copy of the malware in memory. An automatic unpacker can be used in straightforward malware analysis to extract an unpacked copy of the malware. This unpacked malware can be used to create a yara rule for the identification of the malware. In a less straightforward situation, the malware may be able to evade the VM and/or automatic unpacker, and a manual unpacking process may be required. The Volatility Framework assists in locating the malware injection target by identifying open file, open registry keys, and unexpected network connections. When the process containing injected malware is located, the Volatility Framework provides commands to extract the memory allocations associated with the process. One of the memory allocations of an injected process will contain an unpacked copy of the malware. In the case where a yara signature is available for the malware family, a dump of the memory allocations of all processes can be extracted, and the yara rule can be used to identify the processes that contain injected malware.

A.1.5 Debugger Selection

Basic dynamic analysis of malware can be performed by running malware samples and using tools to analyse their execution. This includes tracking process execution with the SysInternals tools [101], analysis of memory dumps using the Volatility Framework, and network traffic analysis using Wireshark. [102]. A deeper level of malware analysis can be performed using a debugger to control the the execution of the sample and to observe its interaction with the operating system. The WinDbg debugger is provided with Debugging Tools For Windows. The Windbg debugger can be used to debug kernel mode and user-mode code, unfortunately Windbg is a complex debugger. Due to operating system improvements, a lot of malware operates in user-mode; this allows malware analysis to be performed using a user-mode debugger. X64dbg is a popular user-mode debugger for 32bit and 64 bit Windows programs [103].

A.1.6 Disassembler Selection

Disassemblers represent compiled programs using the functions contained within the program and the instructions comprising the functions. A description of the disassembly process and the associated challenges is provided by Sikorski et al [92]. Modern disassemblers are interactive tools that support an iterative process of understanding the disassembled program. As the disassembled program's behaviours are understood, the disassembler provides features to capture that understanding, e.g. by assigning a name to a function, naming a global variable, or adding comments explaining the purpose of disassembled code within a function. Modern disassemblers incorporate database technology and propagate naming and type changes throughout the disassembly. The Interactive Disassembler (IDA) [104] was one of the first modern disassemblers, although a free demonstration version is provided, the full product cost may be prohibitive for researchers. Recently a new disassembler called Ghidra [105] has been released as an open source program by the National Security Agency (NSA). Ghidra is free, multi-platform, supports decompilation, disassembly, and a wide range of CPU architectures.

A.1.7 Reverse Engineering Workflow

The process of manual reverse engineering can provide a deep understanding of a program's design and operation. However, reverse engineering can be a time consuming and expensive process. It follows that the goals of reverse engineering should be formulated to avoid unnecessary work. An example of this practice includes jumping over packer code by setting a debugger breakpoint on an API called after completion of unpacking. This avoids time consuming and unnecessary packer analysis. Another example is provided by the identification of complex library functions. For example, the Zlib compression library is readily identified, and the library interface functions can be labelled with the source code function names, while the internal functions can be quickly labelled as Zlib internal functions, this quick identification of library functions can save analysis time. A further example of restricting the scope of reverse engineering is given by the construction of a C2 server emulator for an information stealing malware. The malware behaviours that require reverse engineering are malware initialisation, static configuration decryption, C2 server protocol analysis, and configuration download. Malware functions that do not implement these behaviours do not require analysis.

While some researchers only require a high-level view of malware reverse engineering, we will now provide an overview of the workflow and tools needed for manual reverse engineering. Basic malware analysis includes an examination of program strings, imported functions, and program sections. A quick review of program strings may reveal that the sample is a misidentified benign program. A high-level understanding of a program's function may be obtained by considering the imported functions. Analysis of program sections may reveal a resource section that contains malicious code.

A common scenario occurs when a malware sample is not able to be unpacked by the sendbox or automatic unpacker. This leads to a situation where manual analysis is needed to obtain an unpacked copy of the malware. A manual analysis technique that is widely used involves setting a breakpoint on code following the OEP of the original code. While the address of the OEP in memory is not known ahead of time, it is often possible to guess some common API call that are called in malware initialisation. Candidate API calls for this method are `LoadLibrary`, `GetProcAddress`, `InternetOpen`, and so on. Breakpoints are set on these common API's and the malware samples is executed to see if a break can be obtained either in the latter stages of packing or in the unpacked malware. This process of setting breakpoints on common APIs can be used to brute force packer execution and obtain an unpacked sample. At this stage of a reverse engineering process, web pages from security companies providing some of the reverse engineering details for the malware family can be helpful and may provide snippets of disassembled, unpacked code. Situations also arise where packers contain anti-analysis functions that detect the VM or debugger and require additional reverse engineering time. Manual reverse engineering generally gives good results, but lengthy analysis can be a problem.

A.2 Cythereal Semantics

Malware samples were uploaded to Cythereal and when these samples had been processed, then the unpacked sample and the Cythereal semantics were downloaded. Cythereal semantics consist of disassembled code ('code'), generalised code ('gc'), semantics ('sem') and generalised semantics ('gc'). An example disassembled function is shown in Figure A.2 The format used to store the downloaded Cythereal semantics for this function is shown in Figure A.3.

```
00410371 Core__InitDefaultCallUrlData proc near  ; CODE XREF: DynamicConfig__TryToUpdateBot+80↓p
00410371                                         ; DynamicConfig__Download+31↓p ...
00410371                push    30h ; '0'
00410373                push    0
00410375                push    esi
00410376                call    Mem__Set
0041037B                call    loc_41035D
00410380                mov     eax, lpszAgent
00410385                and     dword ptr [esi+24h], 0
00410389                mov     word ptr [esi], 501h
0041038E                mov     dword ptr [esi+4], 1388h
00410395                mov     dword ptr [esi+28h], 0A00000h
0041039C                mov     [esi+0Ch], eax
0041039F                retn
0041039F Core__InitDefaultCallUrlData endp
```

Figure A.2: Example Disassembled Function

```
{’fn_data’:
    {’fn_hash’: ’ce85e885f9d22851e0947b2298eca49a’,
     ’fn_name’: ’’,
     ’fn_rva’: ’0x10371’,
     ’api_calls’: [],
     ’fn_block_count’: 3},
     ’fn_blocks’:
       [
         {’code’: ["push(’0x30’)", ’push(0)’, ’push(esi)’, "call(’0x7894’)"],
          ’gs’: [’A=B+pre(C)’, ’C=A’, ’memdw(A)=pre(D)’, ’memdw(E+pre(C))=F’,
                 ’memdw(G+pre(C))=H’],
          ’gs_hash’: ’2\xbf@.S\xb6\x05\x14\xc1\r\x13\x98v\x0bmz’,
          ’block_rva’: ’0x10371’,
          ’gc’: [’push(A)’, ’push(B)’, ’push(C)’, ’call’],
          ’sem’: [’esp=A’, ’A= -12+pre(esp)’, ’memdw(A)=pre(esi)’,
                  ’memdw(-8+pre(esp))=0’, "memdw(-4+pre(esp))=’0x30’"],
          ’fn_rva’: ’0x10371’
         },
         {’code’: ["call(’0x1035d’)"],
          ’gs’: [],
          ’gs_hash’: ’\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00’,
          ’block_rva’: ’0x1037b’,
          ’gc’: [’call’],
          ’sem’: [],
          ’fn_rva’: ’0x10371’
         },
         {’code’: ["mov(eax,dptr(’0x229a4’))", "and(dptr(esi+’0x24’),0)",
                   "mov(wptr(esi),’0x501’)", "mov(dptr(esi+4),’0x1388’)",
                   "mov(dptr(esi+’0x28’),’0xa00000’)",
                   ’mov(dptr(esi+12),eax)’, ’retn’],
          ’gs’: [’A=pre(memdw(B))’, ’C=A’, ’memdw(D+pre(E))=F’, ’memdw(G+pre(E))=A’,
                 ’memdw(H+pre(E))=I’, ’memdw(J+pre(E))=K’, ’memw(pre(E))=L’],
          ’gs_hash’: ’\xc8\xdd\xd91^\xc4\xf7\xa6\xf0G\x01w\xbb\xc8J>’,
          ’block_rva’: ’0x10380’,
          ’gc’: [’mov(A,B)’, ’and(C,D)’, ’mov(E,F)’, ’mov(G,H)’, ’mov(I,J)’, ’mov(K,A)’,
                 ’retn’],
          ’sem’: [’eax=A’, "A=pre(memdw(’0x229a4’))", "memdw(4+pre(esi))=’0x1388’",
                  ’memdw(12+pre(esi))=A’, "memdw(’0x24’+pre(esi))=0",
                  "memdw(’0x28’+pre(esi))=’0xa00000’", "memw(pre(esi))=’0x501’"],
          ’fn_rva’: ’0x10371’
         }
       ]
},
```

Figure A.3: Example Cythereal Semantics