Formalization and Detection of Host-Based Code Injection Attacks in the Context of Malware

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Thomas Felix Barabosch

aus

Andernach

Bonn 2018

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Peter Martini, Rheinische Friedrich-Wilhelms-Universität Bonn
2. Gutachter: Prof. Dr. Wim Mees, Königliche Militärakademie Brüssel

Tag der Promotion: 04.09.2018
Erscheinungsjahr: 2018

Summary

The Internet faces an ever increasing flood of malicious software (malware). Threat actors distribute millions of new malware variants every year. They do so for a variety of reasons such as financial gain or political power. The sophistication of malware as well as their target platforms steadily increase. Since a couple of years malware has often utilizes a platform-independent technique called *Host-Based Code Injection Attack* (HBCIA). This attack denotes the local injection of code from an attacker entity into a victim entity. Both entities are usually operating system processes. The malicious code runs within the context of another process, which is contrary to the common belief that each program possesses its own process space. HBCIAs allow the attacker the interception of critical information, escalation of privileges, and covert operation. This trend of conducting local code injections creates a challenge in detecting malware since it blends into the behavior of benign processes. Therefore, this thesis addresses this challenge and elaborates on new ways to detect code injections.

So far, no basic research on HBCIAs in the context of malware has been carried out. There is a lack of understanding in terms of problem definition and problem size. Therefore, we built a model and formally defined HBCIAs. Based on this model, we introduced a taxonomy that allowed us to classify malware according to their algorithms. Then, we showed that almost two thirds of malicious samples of our representative corpus leveraged HBCIAs. This finding implies that local code injections are a relevant problem for security researchers since the detection of HBCIAs implies the detection of a huge share of today's malware. Especially due to the fact that leading operating systems, such as Microsoft Windows, Linux, macOS and Android, are all prone to this attack.

After the problem formalization and problem size estimation, we present two approaches: one static and one dynamic method to detect HBCIAs. Since HBCIAs are a behavior exhibited during execution, it is important to detect its occurrence to prevent further damages. Therefore, our first system *Bee Master* dynamically detects HBCIAs at runtime. It transfers the honeypot paradigm to processes. Its main component the *Queen Bee* observes several child processes called *Worker Bees*. Each *Worker Bee* mocks a possible victim process like *Explorer.exe*. The behavior of each *Worker Bee* is a priori known so that new threads or new memory regions imply an HBCIA. Our approach differs from related work due to its platform-independence, high abstraction level and focus on malware. We implemented and evaluated *Bee Master* for several Windows and Ubuntu Linux versions. The evaluation with several prevalent representatives of HBCIA-employing

malware families as well as many benign programs shows that *Bee Master* reliably detects HBCIAs without false positives.

One major source of information is the memory of victim machines. It reflects the machine's state and in case of an attack it allows us to derive valuable insights about the attacker as well as their hacking tools. However, it is difficult to pinpoint an attack in memory. If malware conducts HBCIAs, it is well hidden in benign processes. Hence, our second system *Quincy* addresses the challenge to statically detect HBCIAs in memory dumps. Its detection heuristic is based on machine learning. We constructed 36 features based on domain knowledge and selected the most appropriate ones. At its core, the detection heuristic leverages a tree-based machine learning algorithm. We evaluated *Quincy* with seven algorithms of which *Extremely Randomized Trees* performed best. Subsequently, we evaluated it on three Windows version with an high quality corpus comprising more than a thousand benign and malicious programs. The results show that *Quincy* improves upon the current state of the art by more than eight percent, when comparing both systems using the ROC AUC score.

Publications

This thesis is mainly based on the following three peer-reviewed publications:

- Thomas Barabosch and Elmar Gerhards-Padilla, *Host-Based Code Injection Attacks: A Popular Technique Used by Malware*, 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE), IEEE, 2014
- Thomas Barabosch, Sebastian Eschweiler and Elmar Gerhards-Padilla, *Bee Master: Detecting Host-Based Code Injection Attacks*, Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Springer International Publishing, 2014
- Thomas Barabosch, Niklas Bergmann, Adrian Dombeck and Elmar Padilla, *Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps*, Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Springer International Publishing, 2017

The following peer-reviewed publications evolved (directly or indirectly) from our research for this thesis:

- Thomas Barabosch, Andre Wichmann, Felix Leder and Elmar Gerhards-Padilla, *Automatic Extraction of Domain Name Generation Algorithms from Current Malware*, NATO Symposium on Information Assurance and Cyber Defense (IST-111), 2012
- Thomas Barabosch, Adrian Dombeck and Elmar Gerhards-Padilla, *ParasiteEx: Disinfecting Parasitic Malware Platform-Independently*, Future Security (FuSec), Fraunhofer Verlag, 2015
- Thomas Barabosch, Adrian Dombeck, Khaled Yakdan and Elmar Gerhards-Padilla, *BotWatcher: Transparent and Generic Botnet Tracking*, Research in Attacks, Intrusions, and Defenses (RAID), Springer International Publishing, 2015
- Thomas Barabosch and Elmar Gerhards-Padilla, *Behavior-Driven Development in Malware Analysis*, The Journal on Cybercrime & Digital Investigations, Volume 1, Number 1, CECyF, 2016

Contents

1 Introduction 1
1.1 Problem Statement . . . 1
1.2 Research Questions . . . 3
1.3 Main Contributions . . . 4
1.3.1 Basic Research on HBCIAs . . . 5
1.3.2 Detection of HBCIAs at Runtime . . . 5
1.3.3 Detection of HBCIAs in Memory Dumps . . . 5
1.4 Roadmap . . . 6

2 Basics 9
2.1 Malware . . . 9
2.2 Malware Analysis . . . 10
2.2.1 Dynamic Analysis . . . 10
2.2.2 Static Analysis . . . 12
2.2.3 Memory Forensic Analysis . . . 13
2.3 Honeypots . . . 15
2.4 Machine Learning . . . 15
2.4.1 Samples and Features . . . 16
2.4.2 The Classification Problem and Supervised Learning . . . 16
2.4.3 Decision Trees . . . 17
2.4.4 Ensemble Learning . . . 18
2.4.5 Forests of Randomized Trees . . . 19
2.5 Conclusion . . . 19

3 Related Work 21
3.1 Formalization and Study of Code Injection Attacks . . . 22
3.2 Prevention of Code Injection Attacks . . . 22
3.2.1 Complication and Prevention of Code Execution . . . 22
3.2.2 Randomization . . . 23
3.2.3 Integrity . . . 27
3.3 Detection of Code Injection Attacks . . . 29
3.3.1 Host-based Approaches . . . 29
3.3.2 Network-based Approaches . . . 33
3.4 Non-Scientific Work . . . 34
3.4.1 Patents Related to Code Injections . . . 34
3.4.2 Security Products Related to Code Injections . . . 35
3.5 Conclusion . . . 36

4 Defining Host-Based Code Injection Attacks 37

4.1 Defining Code Injections . . . 38
4.1.1 Attacker Model . . . 39
4.1.2 Code Injections . . . 39
4.1.3 Host-Based and Remote Code Injections . . . 40
4.1.4 HBCI/RCI vs. HBCIA/RCIA . . . 40
4.2 Advantages and Disadvantages of HBCIAs . . . 41
4.2.1 Advantages of Employing HBCIAs . . . 42
4.2.2 Disadvantages of Employing HBCIAs . . . 44
4.3 HBCIA Algorithms . . . 46
4.3.1 Victim Process Selection Strategy . . . 46
4.3.2 Code Copying . . . 49
4.3.3 Code Execution Strategy . . . 49
4.3.4 An HBCIA Algorithm Taxonomy . . . 53
4.4 The Future of HBCIA-employing Malware . . . 54
4.5 Conclusion . . . 55

5 Measuring Host-Based Code Injection Attacks 57
5.1 Prevalence of HBCIAs in Malware . . . 58
5.1.1 Data Set . . . 59
5.1.2 Methodology . . . 65
5.1.3 Results . . . 66
5.2 Preferred Victim Processes . . . 66
5.2.1 Data Set & Methodology . . . 67
5.2.2 Results . . . 67
5.3 Family Feature HBCIAs . . . 68
5.3.1 Data Set . . . 68
5.3.2 Methodology . . . 71
5.3.3 Results . . . 72
5.4 Prevalence of HBCIA Algorithms . . . 72
5.4.1 Data Set . . . 73
5.4.2 Methodology . . . 74
5.4.3 Results . . . 74
5.5 Conclusion . . . 75

6 Bee Master: Detecting Host-Based Code Injection Attacks at Runtime 77
6.1 Methodology . . . 78
6.1.1 Requirements . . . 78
6.1.2 Overview . . . 80
6.1.3 Queen Bee . . . 81
6.1.3.1 Decision Heuristic . . . 82
6.1.4 Worker Bees . . . 85
6.1.5 Implementation . . . 87
6.1.6 Limitations and Evasions . . . 87
6.2 Evaluation . . . 88
6.2.1 Evaluation Methodology . . . 88
6.2.2 Configuration of the Prototype . . . 89
6.2.3 Data Sets . . . 90

6.2.4 Results . . . 94
6.3 Conclusion . . . 96

7 Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps 99
7.1 Requirements . . . 101
7.2 Methodology . . . 103
7.2.1 Overview . . . 103
7.2.2 Feature Extraction . . . 104
7.2.3 Feature Selection . . . 112
7.2.4 Training and Classification . . . 112
7.2.5 Implementation . . . 114
7.2.6 Discussion of Evasion . . . 115
7.3 Evaluation . . . 115
7.3.1 Data Set . . . 115
7.3.2 Methodology . . . 121
7.3.3 Results . . . 124
7.4 Conclusion . . . 134

8 Applications, Future Work, and Conclusion 137
8.1 Applications . . . 137
8.1.1 Malware Detection . . . 137
8.1.2 Malware Analysis . . . 138
8.1.3 Forensic Analysis . . . 138
8.2 Future Work . . . 139
8.2.1 HBCIAs in General . . . 139
8.2.2 Bee Master . . . 139
8.2.3 Quincy . . . 140
8.3 Conclusion . . . 143

A Sample List of Class Prevalence Estimation 145

B Sample List of Bee Master Evaluation 147

C Sample List of Quincy Evaluation 149

Abbreviations 154

List of Figures 157

List of Tables 159

Bibliography 163

Faber est quisque fortunae suae.

Appius Claudius Caecus

1 Introduction

1.1 Problem Statement

When the Internet became accessible to the public in the 1990s, only some daydreamers imagined how the world would look just ten years later. Digitalization of society was science fiction. Today, just a couple of years later, many aspects of our daily life are digitalized. From our annual tax declaration to video communication with relatives residing in countries far away, the Internet and enabling technologies such as laptops and smartphones are essential to our society, which continues on its way to full digitalization. Indeed, the Western world is on a digitalization highway. We are witnessing how the first generation of real digital natives grow up. This new generation naturally interacts with the enabling technologies such as tablet computers and smart phones. The rapid development of the *Internet of Things* (IoT) results in further digitalized aspects of our life. Ubiquitous objects such as refrigerators, coffee makers or watches start to communicate and interact with each other. Leading manufactures estimate that 50 billion devices will be interconnected in 2020 [1]. This leads to many new opportunities. Not only will entrepreneurs benefit from this trend, but also users will benefit, for example, due to better life quality.

It is utterly important that future devices will be developed with security in mind. Fifty billion devices offer a huge attack surface. However, in the past we witnessed more than once that security is treated as a second-class citizen. When enterprises rush products to the market, just to have the first-mover advantage, security aspects are neglected. In general, devices are only supported for a couple of months. The industry counts on a consume-oriented society that replaces functional devices in short cycles. For instance, Google counts with a 18 month lifetime of Android devices and provides patches only within this period of time [2]. However, this led and still leads to the problem of billions of vulnerable devices.

Such vulnerabilities are the gateway for adversaries. Once they exploited a vulnerability and gained access to a system, they wish, for example, to extract information from the hacked system. This is where malicious software – also known as *malware* – comes into play. This kind of software executes unsolicited on a system and carries out malicious activities such as banking fraud or identity theft. Society faces an ever increasing flood of malware. According to *PandaLabs*, they detected 75 million novel malware samples in 2014 [3] and 84 million in 2015 [4]. This has led to severe economical damages. The antivirus company *Intel Security* estimated that cybercrime has an annual worldwide cost of $400 billion [5]. Industrial countries are heavily affected. For example, Germany has an estimated cost of cybercrime of 1.6% of its gross domestic product (GDP) [5]. Cyber criminals are not the only ones that utilize malware. Whereas cyber criminals mostly tend to enrichment, secret services and armies utilize sophisticated malware for complex operations in the field. This includes, for instance, gathering intelligence [6] and attacking crucial infrastructures like nuclear power plants [7]. It is likely that the problem of malware will not diminish in the near future but rather will receive another boost with the increasing digitalization of our society.

Not only does the flood of malware, its caused financial damage, and its sophistication continue to grow, but also the number of target operating systems increases. *Microsoft* was the undisputed market leader for consumer-oriented operating systems just ten years ago. As of November 2017, Microsoft is still the market leader [8]. However, the tide has turned and old and new competitors are attacking the market position of Microsoft. Other major desktop-oriented (macOS, Linux) or mobile-oriented (Android, iOS) OSes have a significant market share [8]. This also motivates malware authors to write malicious code for these operating systems [9–11].

Nonetheless, malware authors do not reinvent the wheel when writing malware for another operating system. Instead, they rather quickly adapt to other platforms by porting well-known techniques. One of these techniques is the *Host-Based Code Injection Attack*

(HBCIA). In a nutshell, an HBCIA is a local attack from one local entity such as a process or kernel module on another local process. The attacking entity injects code into the victim process. Then, it triggers the code execution within the context of the victim. We will give a precise definition of HBCIAs later (see Chapter 4). This kind of attack comes with a lot of benefits from the attacker's point of view. They include, amongst others, covert operation within a benign process and interception of unencrypted, critical information. We estimate that almost two thirds (see Chapter 5) of malware samples targeting Windows employ HBCIAs. A wide range of malware families such as banking Trojans, ransomware, or computer worms uses them. Moreover, many malware families that participate in sophisticated targeted attacks also employ HBCIAs (e.g. [6, 12, 13]).

We believe that a thorough investigation of this phenomenon and the proposal of detection approaches leads to a higher detection rate of malware and increased Internet security in general. Therefore, this thesis focuses on the topic of *Host-Based Code Injection Attacks* in the context of malware. It lays a solid foundation with its basic research and analysis as well as proposes two approaches to – dynamically and statically – detect HBCIAs on various operating systems.

1.2 Research Questions

We have touched on the problem of malware and *Host-Based Code Injection Attacks* in our initial problem statement. Malware and closely related to it *Host-Based Code Injection Attacks* are ubiquitous problems. Take for example the number of Internet users in 2015. More than 80% in the developed world and already millions of the undeveloped world had daily access to the Internet and computing devices in general. Nevertheless, there were still four billion people in developing countries that remained offline [14]. Therefore, many of us face or will face malware and HBCIAs, whether be it knowingly or unknowingly.

During our work as malware analysts, we faced HBCIAs in a plethora of malware families. Moreover, most malware analysis reports discuss the utilized code injection scheme, be it an injection by the malware packer or by the unpacked malware. Our goal is to investigate this phenomenon in depth and to develop detection mechanisms. In this section, we pose the two main questions – analysis and detection of HBCIAs – that drove our research throughout the last years. These two questions lead us to the main contributions of this thesis (see Section 1.3).

> How does modern malware employ Host-Based Code Injection Attacks, what are the consequences, and what is the problem size?

The first research question covers the basics of HBCIAs. While related research mostly focused on code injection attacks in general, we are the first to discuss HBCIAs in the context of malware in detail (see Chapter 3). In contrast to classical code injection attacks, malware often employs techniques that the operating system provides. We published a seminal paper on this research question [15] and we will address this topic with great detail in the first part of our thesis (see Chapters 4 and 5).

> How can we detect this malicious behavior using dynamic and static analysis techniques?

The second research question focuses on the detection of this particular behavior. While dynamic detection protects users at runtime, static detection is of great value to malware and forensic analysts. We published two seminal papers on this research question ([16] and [17]) and we will address this topic in detail in the second part of our thesis (see Chapters 6 and 7).

1.3 Main Contributions

In this section, we summarize the main contributions of this thesis. First, we conducted basic research on *Host-Based Code Injection Attacks* that defines and classifies this kind of attack in the context of malware. Second, we proposed a system to detect HBCIAs at runtime. Its main objective is live user protection. Third, we proposed a system that statically detects HBCIAs in memory dumps utilizing machine learning techniques. Its main goal is to improve upon current systems to aid forensic and malware analysts. We will examine in detail these three main contributions in the following sections. The main contributions of this thesis were published in several peer-reviewed seminal papers:

- *Host-Based Code Injection Attacks: A popular technique used by malware* [15]
- *Bee Master: Detecting Host-Based Code Injection Attacks* [16]
- *Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps* [17]

Please note that this thesis goes further than these individual publications. It contextualizes our work by discussing HBCIAs in the context of malware, the need to dynamically and statically detect them, and their future implications. Furthermore, we improved the proposed systems and evaluated them again to increase the insights on the topic of HBCIAs.

1.3.1 Basic Research on HBCIAs

Even though the malware analysis community faces HBCIAs every day, no basic research on this phenomenon in the context of malware has been carried out so far. We are the first to investigate this in detail.

Our basic research defined this kind of attack and allows further systematic research on this topic. Since HBCIAs are based on a platform agnostic concept, cross-platform detection of malware can be achieved by focusing on this technique rather than on the individual operating system-dependent malware families. Furthermore, we showed in several measurements with current malware that, for example, HBCIAs are very prevalent (almost two thirds of current malware employs this technique) or that these attacks are a family feature, i.e. it can be found in all members of a malware family (variants and versions). These measurements emphasize the need to investigate HBCIAs in particular.

1.3.2 Detection of HBCIAs at Runtime

There are approaches that focus on detecting HBCIAs at runtime. However, they are impractical and slow (e.g. [18, 19]), they are too tightly coupled to operating system-specific knowledge (e.g. [20–22]), or they require new hardware (e.g. [23, 24]). Also, given the fact that HBCIAs are a platform- and operating system-agnostic problem, most solutions focus on one platform and operating system.

We are the first to present a platform and operating system agnostic approach called *Bee Master* that detects HBCIAs at runtime. Our approach is neither based on domain-specific knowledge nor does it require additional hardware. Rather it employs the honeypot paradigm to operating system processes and offers a set of victim processes to the malware. We evaluated *Bee Master* with 38 malware families and several hundred benign programs on Windows and Linux. The evaluation shows that it detected all HBCIAs without having any false positive.

1.3.3 Detection of HBCIAs in Memory Dumps

The detection of HBCIAs in memory dumps is an everyday task of forensic and malware analysts. Current approaches (e.g. [25–27]) come with high false positive rates and are not scientifically grounded or have a too coarse detection granularity (e.g. [28]).

We present a scientifically based approach for statically detecting HBCIAs in memory dumps. Our approach *Quincy* detects them by leveraging supervised machine learning

techniques. We employ up to 36 unique features that are common to HBCIAs. The result is a method working with modern Windows operating systems from Windows XP to Windows 10, which we released as a *Volatility* plugin on *github* [29]. During our investigation on HBCIAs in memory dumps, we gathered the most comprehensive data set of representatives of HBCIA-employing malware families that is available today. We created *YARA* signatures for each family to verify a successful infection and to ensure a precise ground truth. We published this data set on github as well [30]. A comparison to the current state of the art *Malfind* [25] showed that *Quincy* had more true positives (of up to 27% on Windows 7) and less false positives (of up to 63% on Windows XP). This statistically significant improvement yielded an increase of the area under the receiver operating characteristic (*ROC*) curve of up to 8% (on Windows 7).

1.4 Roadmap

This thesis is divided into eight chapters that cover the topic *Host-Based Code Injection Attacks* and their detection in-depth.

In **Chapter 2**, we present the preliminaries that are fundamental to understand this thesis. Since this thesis focuses on HBCIAs in the context of malware, we provide an overview of malicious software and its analysis. To understand our static and dynamic detection approaches, the reader requires basic knowledge of honeypots and machine learning, which we discuss also in this chapter.

Chapter 3 reviews related work and arranges our work in the scientific landscape. Major fields that are related to our thesis are the prevention and the detection of code injection attacks.

Chapter 4 discusses the fundamentals of HBCIAs in detail. At first, we present a model that allows us to define them. This model assumes general concepts of the *Von Neumann* architecture and general concepts of multi-tasking operating systems like Windows and Linux. Then, we take a close look at the fundamentals of HBCIA algorithms that current malware employs. This leads to a taxonomy that allows us to classify current malware families according to their HBCIA algorithm. Finally, we discuss future implications of HBCIAs.

Chapter 5 measures the impact of HBCIAs in practice. At first, we estimate the prevalence of HBCIAs in current malware and the preferred victim processes. Then, we show that an HBCIA is a malware family feature that does not change either between different versions or different variants of a malware family. Finally, we scrutinize the different HBCIA algorithms that we have defined in the previous chapter.

In **Chapter 6**, we present a system for detecting HBCIAs at runtime. The system called *Bee Master* applies the honeypot paradigm to operating system processes. It relies only on concepts that are common to all current multi-tasking operating systems such as Windows and Linux. *Bee Master* detects HBCIAs platform-independently as proven in an evaluation on several versions of Windows and Ubuntu Linux with prevalent malware families, artificial samples and goodware.

Chapter 7 proposes a system to detect HBCIAs in memory dumps. Our approach *Quincy* is based on similar concepts as *Bee Master*. It employs machine learning techniques in order to detect HBCIAs in memory dumps. We evaluated *Quincy* on several versions of Windows and compared it to the current state of the art *Malfind* [25] as well as *Hollowfind* [27].

In **Chapter 8**, we discuss direct applications of our thesis, future work, and conclude this thesis.

If I have seen further, it is by standing on the shoulders of giants.

Sir Isaac Newton

2 Basics

This chapter discusses the basics that are required to follow this thesis. At first, we discuss malware in general and how its analysis is conducted. Since our thesis focuses on malware, it is important to have a basic idea of malware and to understand how malware is analyzed today. We then shift our focus in the next section and present honeypots. A basic understanding of honeypots is required since our system *Bee Master* (see Chapter 6) transfers the honeypot paradigm to operating system processes to dynamically detect HBCIAs. Next, we present the fundamentals of machine learning that are relevant to our thesis. Our second detection approach *Quincy* (see Chapter 7) utilizes machine learning techniques. Finally, we recapitulate the main ideas of this chapter in a concluding section.

2.1 Malware

The term *malware* is a blend of the two terms **mal**icious and soft**ware**. There are several definitions, for example, the English dictionary *Meriam-Webster* defines *malware* as "software designed to interfere with a computer's normal functioning" [31]. A more formal definition is given by Kramer et al.. They define malware as "A software system s is malware by definition if and only if s damages non-damaging software systems (the civil

population so to say) or software systems that damage malware (the anti-terror force so to say)." [32]. The utilization schemes of malware are manifold. Cyber criminals may utilize malware for confidential information theft [6], banking credential theft [33], attacking third-parties [34], cyberwarfare [7], or resource theft [35].

Today, on one side there are criminal gangs that target the general public due to monetary gains, e.g. with banking Trojans like *Tinba* [36], *Rovnix* [37] and *Zeus* [38]. Several actors offer their services – commonly known as *Malware as a Service* (MaaS) – in order to help others to commit cyber crimes [39]. Moore et al. estimated that these criminal gangs cause damages worth several billions of dollars each year [40].

On the other side, there are secret services that supposedly target nations, dissidents, or non-governmental organizations due to intelligence. Two prominent cases recently gained media attention. First, the case of *Regin*, where the secret services *NSA* and *GCHQ* allegedly employed this sophisticated espionage kit to spy on, amongst others, the European Union [41, 42]. Second, the Syrian civil war, where the belligerents allegedly spied on each other with mostly simple Trojans [43, 44].

2.2 Malware Analysis

Malware analysis denotes analyzing a malicious program with the objective to gain understanding of its internals. Classically, malware analysis is divided into dynamic analysis and static analysis. Whereas malware executes during dynamic analysis, this is not the case during static analysis. Our first detection system *Bee Master* that we present in Chapter 6 is a dynamic one. Lately, a special form of static analysis has emerged: memory forensics. As a consequence of the rapid development of memory forensic frameworks, malware analysts conduct more and more memory forensic analyses. Our second detection system *Quincy*, which we introduce in Chapter 7 conducts memory forensic in an automated fashion.

2.2.1 Dynamic Analysis

During dynamic analysis of malware, malware analysts execute the malware in a controlled environment. This permits them to observe the malware's behavior and its interactions with the environment at runtime, yielding valuable information about the malware and its internal states.

However, there are two major drawbacks of dynamic analysis. The first one is that not every possible program path is executed [45]. There might be dormant behaviors

that the malware only exhibits when certain conditions are met. An example for a dormant behavior was the time-based update mechanism of *Conficker.C* [46]. For analyzing dormant behaviors, analysts have to statically analyze the malware to explore all its program paths. Another drawback that becomes increasingly more severe is *evasive malware* [47]. Malware authors utilize dynamic analysis evasion to complicate the analysis of their software. They employ several techniques to evade dynamic analysis: i.) some of them (e.g. virtual machine detection [48]) target automatic analysis systems like sandboxes, ii.) some of them (e.g. debugger detection [49]) target the manual analysis carried out by human malware analysts.

Dynamic Analysis Techniques

The simplest form of dynamic analysis is called *Black Boxing* [50]. This technique describes the process of executing a malicious program and observing its interactions with its environment *without* peeking into the program's code or its internal states. Black boxing can be applied at host-level, at network-level, or in a combination of both. For instance, at host-level, a malware analyst may be interested in new processes or files that the malware creates as well as the modifications of system configurations. At network-level, a malware analyst may be interested in the hosts that the malware connects to and the data that it exfiltrates.

Typically, *Virtual Machines* (VMs) are utilized when analyzing malware. VMs emulate computer systems [51]. From a malware analyst's point of view, they offer a broad range of advantages like revertable system states.

There are also special systems for dynamic analysis automation called *sandboxes* [52]. They conduct behavioral analysis of malware samples in an automated fashion. A sandbox executes a malware sample for a couple of minutes, observes the behavior of the sample, and finally outputs a report. To the best of our knowledge, there is no evaluation of the optimal runtime. In practice sandboxes grant malware typically between two and five minutes to run. Most of these systems are based on VMs but there are also systems based on bare-metal machines [53].

Developers utilizes *debuggers* to find and fix errors in their programs [54]. Debuggers allow them, for instance, to step through the code instruction-wise, to execute the code until it reaches a certain point or it fulfills a certain condition, as well as to inspect the registers or the memory of a program. They are a valuable tool for malware analysts, since they permit them to inspect the malicious code while it is interacting with its environment.

2.2.2 Static Analysis

During static analysis the malware does not execute. Although, no execution context is available during static analysis, the whole program can be inspected including all possible program paths [45].

However, there are two major drawbacks associated with static analysis. First, the majority of today's malware samples is packed, i.e. the original code is compressed or encrypted. It is then decompressed or decrypted at runtime [55]. This renders a sole static analysis of malware impracticable. Malware analysts must unpack the malware sample before they can conduct a proper static analysis. Second, the code of malicious binaries is often obfuscated (e.g. [56–58]), which further impedes an analysis and often demands the time-consuming implementation of custom deobfuscation scripts.

Static Analysis Techniques

Basic static analysis deals with a malware sample without taking its code into account. For example, the inspection of a malware sample's header is a static analysis technique. Cryptographic hash functions such as *MD5* or *SHA256* are employed to fingerprint malware samples [59]. If the hash value of the sample is known because it had already been analyzed, then no further analysis is required. Unfortunately, cryptographic hash functions are prone to minimal modifications, which result in completely different hash sums. Therefore, other static similarity measures were proposed including *Fuzzy Hashes* [60], *Pehash* [61] and *Import Hashing* [62]. Further basic static analysis techniques determine if a malware sample is packed. Several publications focus on executable packer detection (e.g. [55, 63]) and consequently on static unpacking (e.g. [64]).

Disassembling

We assume in the following that the malware analyst has no access to the malware's source code and that the malware at hand comes in binary form. This is a reasonable assumption for the great majority of malware samples since distributing malware in binary form increases its analysis complexity. A primitive analysis approach would be analyzing the malware's machine code, for example, as hexadecimal dump in a hexeditor. Listing 2.1 shows the machine code of a *Rovnix* function as hexadecimal dump.

```
55 8B EC 51 C7 45 FC 00
00 00 00 83 7D 0C 00 74
08 8B 45 0C 3B 45 10 76
```

```
07 C7 45 FC 57 00 07 80
8B 45 FC 8B E5 5D C2 0C
```

LISTING 2.1: Hexadecimal representation of x86 machine code of Rovnix (MD5: *284c8188657cabad50c3192200ea445a*)

Only very skilled and experienced analysts understand such a hexadecimal representation. For instance, *0x55* is the encoding of the instruction *push ebp* and *0xEC8B* (due to the endianness of Intel x86) is the encoding of the instruction *mov ebp, esp*. *Disassemblers* decode the binary to assembly code. Machine code is the code that a micro processor understands. A disassembler decodes – or more formally speaking disassembles – the machine code of Listing 2.1 to the human-understandable assembly code in Listing 2.2.

```
.text:760011A0 sub_760011A0 proc near
.text:760011A0 var_4= dword ptr -4
.text:760011A0 arg_4= dword ptr  0Ch
.text:760011A0 arg_8= dword ptr  10h
.text:760011A0
.text:760011A0 push    ebp
.text:760011A1 mov     ebp, esp
.text:760011A3 push    ecx
.text:760011A4 mov     [ebp+var_4], 0
.text:760011AB cmp     [ebp+arg_4], 0
.text:760011AF jz      short loc_760011B9
.text:760011B1 mov     eax, [ebp+arg_4]
.text:760011B4 cmp     eax, [ebp+arg_8]
.text:760011B7 jbe     short loc_760011C0
.text:760011B9
.text:760011B9 loc_760011B9:
.text:760011B9 mov     [ebp+var_4], 80070057h
.text:760011C0
.text:760011C0 loc_760011C0:
.text:760011C0 mov     eax, [ebp+var_4]
.text:760011C3 mov     esp, ebp
.text:760011C5 pop     ebp
.text:760011C6 retn    0Ch
.text:760011C6 sub_760011A0 endp
```

LISTING 2.2: Disassembled function of Rovnix using IDA Pro [65]

The disassembled code exhibits increased readability compared to the hexadecimal code. Typically, malware analysts analyze malicious code at this level of abstraction.

2.2.3 Memory Forensic Analysis

So far, we have discussed the two traditional branches of malware analysis. Now, we shift our focus to a recent branch. Our second approach *Quincy* that we will present in

Chapter 7 solves a problem that malware analysts but also forensic analysts face daily: Detecting a malicious binary that has injected itself into another process space. Once they have detected this binary, they can continue with its detailed analysis. To solve this problem, we utilize *memory forensic analysis*, which has become popular in the context of malware throughout the last years, owing to the rapid development of memory forensic frameworks.

Memory forensic analysis is a subcategory of *Computer Forensics*, which is the science of identifying, preserving, and examining evidence on computer systems, all done in a forensic-sound fashion [66]. Computer forensics is a broad field that includes areas like storage media forensics and memory forensics. However, memory forensic analysis in the context of malware analysis gained its momentum only a couple of years ago. This is a consequence of the easy and free access to frameworks such as *Volatility* [25] and *Rekall* [67].

We consider memory forensic analysis in the context of malware as a combined approach of dynamic and static analysis. The malware is not executed during the analysis (static analysis) because the forensic analyst analyzes a memory dump that shows the state of a system at point in time t. However, the malware was executed in the environment at hand before the memory dump was taken. Therefore, its interactions are observable (dynamic analysis). This definition may differ from the literature. Where the literature is written from the point of view of forensic analysts, this thesis is written from the point of view of malware analysts.

A memory dump is an image of a computer's volatile main memory at point in time t. It comprises a wealth of data. For example, it contains the list of running processes of the operating system or private keys of hard disk encryption tools. A plain image of a computer's volatile main memory is just a huge binary string. In this form it is of no use to analysts. First, they have to overcome the *semantic gap* to conduct an analysis of the system's state at point in time t [68]. The semantic gap denominates the fact that there is a gap between how the lower layers interpret the memory dump data (low semantic value) and how the operating system interprets it (high semantic value) [69]. Several seminal papers have investigated this problem (e.g. [70, 71]). In practice, analysts have access to mature frameworks like *Volatility* [25] to overcome the semantic gap.

They are popular, especially in the malware analysis community. Malware analysts execute a sample in a VM and after a couple of minutes they create a memory dump of this VM. The hypervisor usually offers the functionality to take a consistent memory dump, e.g. with *dumpvmcore* in *VirtualBox* [72]. Then, they analyze the behavior of the malware by analyzing the forensic artifacts with the help of such frameworks. The advantage of this approach is that analysts are not restricted to neither kernel space nor

user space and that they have access to all information that was available at point in time t. Executing a sample and observing the results of its interactions with the environment in a memory dump is similar to blackboxing, which is a dynamic analysis method (see Section 2.2.1).

2.3 Honeypots

Spitzner defines a *honeypot* as "*a security resource whose value lies in being probed, attacked or compromised*" [73]. These resources allow the collection of data on attack patterns, the detection of previously unknown attack vectors, or the employment to distract attackers. Possible application areas of honeypots in the security domain are manifold. We therefore introduce honeypots only in the context of malware. Please note that there are other applications such as detecting password cracking [74] or profiling attacker behavior [75] that are not directly related to malware.

Depending on their application, honeypots can be either host-based (e.g. *Ghost* [76]) or network-based (e.g *Nepenthes* [59]). They can be either a client (e.g. *Thug* [77]), i.e. searching to be attacked, or they can be a server (e.g. *honeyd* [78]), i.e. waiting for an attack. Furthermore, the interaction level between the honeypot and the attacker is an important success factor. Interaction levels stretch from low interaction to high interaction. Whereas low interaction honeypots simulate attackable entities (e.g. *dionaea* [79]), high interaction honeypots offer real entities (e.g. *CaptureHPC* [80]). The higher the interaction, the more attack information can be extracted. However, the higher the interaction, the higher the probability of misuse since services are not simulated.

2.4 Machine Learning

Machine learning evolved from the field of artificial intelligence. In a nutshell, machine learning is the science of learning from data and making predictions based on it [81].

In the following sections, we introduce the basics of machine learning relevant to this thesis. At first, we define samples and features. Then, we introduce the classification problem and supervised machine learning. Our system *Quincy* (see Chapter 7) solves a problem of this class since it classifies memory artifacts as benign or malicious. It solves it by utilizing supervised machine learning. Finally, we present the machine learning algorithm of *Quincy*'s final model. We derive *Extremely Randomized Trees* from simple *Decision Trees* and *Ensemble Learning*. Even though we evaluated our system with several

algorithms in Section 7.3, we describe only the best performing algorithm *Extremely Randomized Trees* in the following. The reader should consult introductory literature on machine learning such as Russel et al. [81] or Flach [82] for explanations of the other machine learning algorithms utilized in Section 7.3.

2.4.1 Samples and Features

We define a sample s as single entity of a specific class. For instance, samples of the vehicle class are *Freightliner M2 dump truck* and *BMW X3 G01*. Features describe measurable aspects of samples [82]. For instance, a vehicle has, amongst others, a top speed and a tank capacity. A feature f assigns to each sample s a real number $f(s)$. Given a fixed set of features $F = \{f_1, ..., f_n\}, n \in \mathbb{N}$, we can see a sample s as an n-dimensional vector of features $(f_1(s), ..., f_n(s)) \in \mathbb{R}^n$. This identifies samples with points in a vector space, called *feature space*. The identification naturally defines a notion of distance and similarity between samples. For instance, sports cars should be clustered closer to each other than to trucks and golf cars because their top speed is much higher. Many classification algorithms are based on geometric properties of samples in the feature space, e.g. SVM [83] uses a hyperplane to split the feature space in two half-spaces and KNN [84] uses distance in the feature space to assign samples to a cluster.

2.4.2 The Classification Problem and Supervised Learning

Given a set of predefined classes $C_0, C_1, \ldots, C_n$, the problem of assigning a sample s to one of the classes C_i is called *classification problem* [85]. The simplest classification problem is binary classification. Either a sample belongs to class C_0 or it belongs to class C_1. For instance, the detection of malware can be modeled by two classes $C_{goodware}$ and $C_{malware}$ that need to be distinguished.

One approach to solve classification problems is *Supervised Machine Learning*. This term describes learning *with* a teacher. More formally, it describes the problem of inferring a function from a set of labeled training data [81]. The labeled training data is a set of pairs, consisting of known samples together with their correct classification. A supervised learning algorithm induces a function based on this data with the objective to correctly classify unseen data.

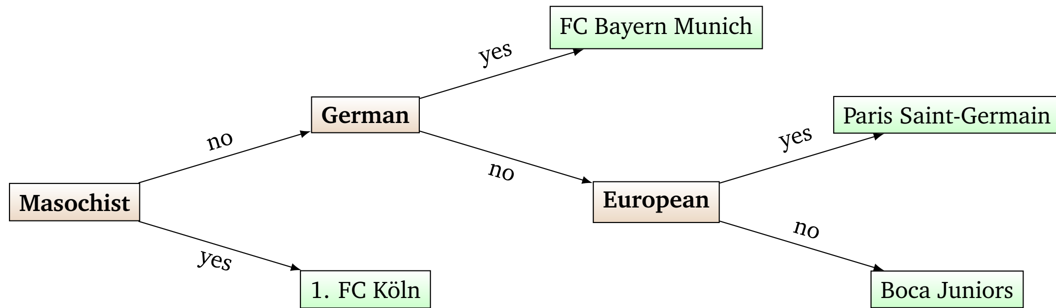


FIGURE 2.1: Decision tree for choosing the right football club. Brown nodes represent tests and green leafs represent decisions.

2.4.3 Decision Trees

Quincy's model selection showed that the algorithm *Extremely Randomized Trees* performs best on our problem (see Section 7.3). In the following three sections, we derive this algorithm using a bottom-up approach. First, we introduce *Decision Trees* in this section. They are the smallest constituent unit of *Extremely Randomized Trees*. This algorithm assembles many of these units to one greater structure by utilizing *Ensemble Learning*, which we explain in the next section. Finally, we assemble the concepts of this and the next section to *Extremely Randomized Trees* in the section after next.

Decision Tree is a supervised machine learning algorithm that partitions the feature space into a set of rectangles [86]. It is based on a tree data structure to make decisions. A sample traverses this tree starting at its root and a series of tests is performed. Each non-leaf node represents a test and each leaf a decision to classify this sample. Figure 2.1 illustrates a *Decision Tree* for choosing the right football club. Each brown node tests a feature, e.g. if the future fan is interested in European football. The green leaves suggest a football club, e.g. the Argentinian club Boca Juniors.

The de-facto standard algorithm for *Decision Tree* generation is *CART* (Classification And Regression Trees) proposed by Breiman et al. in 1984 [87]. *CART* is a greedy algorithm. It chooses at each node of the tree how to split the current set of samples into subsets based on their features.

Decision Trees tend to reflect anomalies and outliers of the training data [88]. A method to address this shortcoming is *Tree Pruning*. Pruned trees lose a lot of their complexity. In a nutshell, tree pruning algorithms reduce the impact of noise by removing the least reliable branches of the tree and by combining leaves.

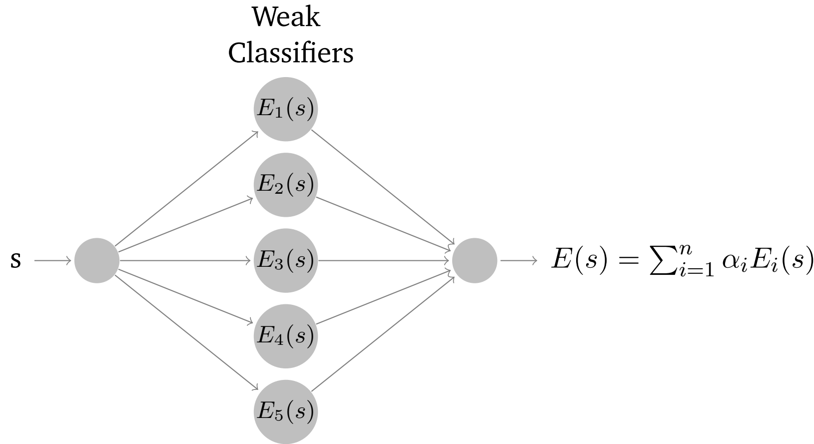


FIGURE 2.2: Predicting a class using an ensemble classifier E with five weak classifiers E_i, where $i \in \{1, .., 5\}$.

2.4.4 Ensemble Learning

The preceding section has stated that simple *Decision Trees* tend to reflect anomalies and outliers of the training data. The bias-variance trade-off expresses this dilemma. It denotes the problem of minimizing two main roots of errors in supervised machine learning algorithms that prevent an algorithm from generalizing [82]. The bias describes the difference between the expected predictions and the true predictions. It is responsible for underfitting. The variance describes the variability of predictions. It is responsible for overfitting. One way to counter this problem in *Decision Trees* is to prune them. Another way to tackle this problem is *Ensemble Learning*. Ensemble learning combines multiple (weak) classifiers to one ensemble classifier. The predictive performance of such an ensemble is often better than the performance of each weak classifier [85]. This is based on statistical intuition: Conducting several measurements and then averaging them leads to more stable and reliable estimates [82]. However, the algorithmic complexity and model complexity increase as well. Ensembles types are, for instance, *Bagging*, *Tree Forests*, and *Boosting* [88].

Figure 2.2 illustrates how a learning ensemble E predicts a class for sample s. Each weak classifier predicts $E_i(s)$, where $i \in \{1, ..., n\}$ and $n \in \mathbb{N}$ is the number of classifiers. The prediction is given by averaging the weighted predictions of the weak classifiers.

Even though the following ensemble learning algorithms work with different types of weak classifiers, they are typically utilized in conjunction with *Decision Trees* in practice [82, 88]. Therefore, we assume the weak classifiers to be standard *(CART) Decision Trees* in the following.

2.4.5 Forests of Randomized Trees

Forests of Randomized Trees is an ensemble method that consists of several randomized *Decision Trees*. The introduction of randomization to the ensemble learning process yields a decrease of the model's variance with a typical increase of the model's bias [82]. The two most popular members are *Random Forest* [89] and *Extremely Randomized Trees* [90].

Extremely Randomized Trees proposed by Geurts et al. [90] heavily rely on randomization. They are based on the idea of *Bootstrap Aggregation*, also known as *Bagging*. The main idea of *Bagging* is drawing several subsamples of the original set and train one weak classifier of the learning ensemble on each subsample [82]. The sampling method is done by uniformly drawing samples with replacement from the original sample set. Since samples are drawn with replacement, a bagged set may contain duplicates. Training the weak classifiers on different bootstrapped sets creates diversity, which in turn increases the robustness of the ensemble.

Another technique called *Subspace Sampling* is utilized to further increase variation. It induces each *Decision Tree* from a different random subset of the feature set, i.e. the split that is chosen is not the best split among all features but rather the best split among a random subset of them [82]. In case of *Extremely Randomized Trees*, a random cutpoint for a feature $f \in F$ is uniformly drawn to further increase randomization. At each node, it creates K splits with K being a parameter of the algorithm (a typical default choice $K = \sqrt{|F|}$). Therefore, it examines K features at each node. For each feature $f_k, 0 \leqslant k \leqslant K$, it uniformly draws a cutpoint from $[f_{min}, f_{max}]$, where f_{min} denotes the minimal value of f regarding the current set and f_{max} denotes the maximal value of f regarding the current set. Then, it takes the split that yields the highest score with respect to a scoring metric, e.g. the precision, the recall, or the ROC AUC score.

2.5 Conclusion

This chapter has discussed the fundamentals of this thesis. Since this thesis focuses on *Host-Based Code Injection Attacks* in the context of malware, we have given an introduction to malware and its analysis first. Our two systems to detect HBCIAs employ dynamic analysis (*Bee Master* in Chapter 6) and forensic analysis (*Quincy* in Chapter 7). *Bee Master* incorporates the honeypot paradigm. Therefore, we have discussed honeypots including high-interaction and host-based systems. *Quincy* relies on machine learning techniques. We have presented *supervised* machine learning and in particular the

algorithm *Extremely Randomized Trees* that achieved the best performance in *Quincy*'s evaluation in Section 7.3.

Plus ça change, plus c'est la même chose

Jean-Baptiste Alphonse Karr

3 Related Work

This chapter plants our research in the research landscape by discussing related work, which covers three areas: The first area is the formalization and study of code injection attacks in general. The second area is the prevention of *Host-Based Code Injection Attacks* (HBCIAs). There have been numerous publications in this area that mostly aimed at the prevention of exploitation due to software bugs, e.g. stack-based buffer overflows. This topic was intensively researched in the early 2000s concomitant with the massive outbreaks of computer worms. The third area is the detection of code injection attacks. In this field, many publications have focused on the network-based detection of code injection attacks. However, there have been few publications that have dealt with the continuously increasing threat of HBCIAs. In addition to related scientific work, we surveyed related software patents and products. Our field is a practical one and the computer security industry significantly invests into the development of mitigation techniques.

Since our thesis focuses on binary code injections carried out by malware, we did not survey other forms of injection techniques like *SQL injections* [91] or *cross-site scripting* (XSS) [92] in the following sections. These techniques are out of scope.

3.1 Formalization and Study of Code Injection Attacks

The formalization and study of code injection attacks is a prerequisite for further studies in this field. Younan et al. [93] surveyed common techniques for injecting code into C and C++ programs. They distinguished between indirect attacks (indirect pointer overwriting and exploiting of heap-based overflows, dangling pointer references, as well as integer errors), exploiting data-based overflows, as well as attacks on format string vulnerabilities. The origin of code injection attacks can be either host-based or network-based. Ma et al. [94] as well as Polychronakis et al. [95] studied network-based code injection attacks in detail.

Whereas the presented publications focused on code injections that only exploit programmatic errors (e.g. dangling pointers), we focus on HBCIAs in the context of malware, which also injects its code with the help of interfaces offered by the operating system. Malware may also use programmatic errors for injection, e.g. the espionage malware *Stuxnet* [7]. We are the first to introduce a formalization of HBCIAs in the context of malware and to classify the various modus operandi. Since our formalization is more abstract, it is agnostic to the underlying technical injection method. This allows us to discuss HBCIAs on different operating systems (e.g. Windows, Linux, macOS) and architectures (e.g. x86, x64, ARM).

3.2 Prevention of Code Injection Attacks

This section discusses related work concerning the prevention of code injection attacks. Even though these approaches focus on the prevention, they may also allow the detection of code injection attacks – Be it explicit, like preventing the execution of a payload at a certain address or be it implicit, like an access violation due to a process crash.

3.2.1 Complication and Prevention of Code Execution

From a defender's point of view, the main objective is blocking the execution of injected code to prevent it from doing any harm. If this is neither possible nor feasible then the defender should at least raise the bar for attackers and complicate the execution of injected code.

The beginning of the 2000s was the time of major computer worm outbreaks. These worms spread mainly via buffer overflows. Back then, buffer overflows were considered as "the vulnerability of the decade" [96] and constituted a major topic in computer

security research. A popular area of research focused on the prevention of buffer overflows (e.g. [97–101]). One notable defense was the introduction of the non-executable stack [102]. Salamat et al. ([103, 104]) proposed multi-variant program execution to detect and prevent buffer overflows. They executed several slightly different instances of the same program in parallel, which performed equivalent computations. Their model assumed that buffer overflows may occur and it detected them by deviations in one or more instances. Another line of research focused on the prevention of unexpected system calls (e.g. [105, 106]). Such system calls might emerge from injected code.

In order to thwart techniques like code-signing and non-executable memory, *return-oriented programming* (*ROP*) was invented ([107–110]). *ROP* allows an attacker to hijack the control flow of a program by only controlling its call stack. Several defenses against *ROP* have been proposed (e.g. [111–114]).

HBCIAs carried out by malware are different. Whereas the discussed seminal papers detect the execution of exploits, they may not detect HBCIAs. This is due to the fact that HBCIAs are mostly carried out with the help of regular system calls. Our approach *Bee Master* is independent of the techniques utilized. It detects, on the one hand, HBCIAs that employ regular system calls. On the other hand, it detects HBCIAs using the above-mentioned attack methods. This is owed to the fact that after such an attack the malware, for example, loads detectable modules, which diverges form the expected behavior.

3.2.2 Randomization

Randomization of several computer system components have been the foundation of many ideas such as *Data Layout Randomization* and *System Call Randomization*. Randomization of several aspects of a computer system may lead to execution failures of injected code. This is due to assumptions the injected code may take (e.g. the API call *VirtualAlloc* resides at address *0x00767689*) or the fact that it may not encounter what it is looking for (e.g. the API call *VirtualAlloc* in general).

Whereas many different approaches were proposed for preventing code injection attacks, only a few of them, like *Address Space Layout Randomization* (ASLR) [115], found their way into major operating systems such as Microsoft Windows and Linux. The reason for this may be the demand of major changes to the kernel. Therefore, many vendors shunned promising approaches like *Instruction Set Randomization* (ISR) [116] because of their impracticality.

In the following sections, we survey proposals that randomize the data layout, the instructions, the system calls, or the operating system itself.

Data Layout

One successful class of countermeasures are *Data Layout Randomizations*. They were introduced between the late 1990s and the early 2000s. For example, injected code assumed back then that certain system libraries would always be mapped to certain addresses. Randomization of data layouts, however, did not lead to the end of code injections. It just raised the bar by reducing the reliability of exploits.

The PAX project [117] introduced several countermeasures against code injection attacks. These included *Non-Executable Pages*, i.e. pages can be marked as non-executable resulting in an exception if executed, and *Address Space Layout Randomization* (ASLR), i.e. randomly relocating code and data pages. Many of the leading operating systems like iOS and Microsoft Windows (DEP [118]) incorporated the *NX-bit* as an implementation of non-executable pages. Xu et al. [119] proposed a system for *Transparent Runtime Randomization*. It randomly relocated the heap, the stack, and shared libraries at runtime. Bhatkar et al. ([120, 121]) presented *Address Obfuscation*, which was based on address randomization. They also added features like the permutation of the variable order. This even further decreased the probability of a successful exploitation. Kil et al. [122] presented *Address Space Layout Permutation* (ASLP). They criticized that ASLR introduced insufficient randomization and that similar approaches (e.g. [120, 121]) required access to the application's source code. ALSP was a binary rewriting system that modified the locations of static code and data to a random location. In addition, it performed fine-grained permutations of the functions and the data in the binary. Bhatkar et al. [123] introduced *Data Space Randomization*. Their proposal randomized the representation of data objects in programs. Thus, their approach protected against corruption of *non-control flow data*. However, it was limited to source code. Giuffrida et al. [124] proposed *Fine-Grained Address Space Randomization*, which allowed ASLR on a finer-grained level in user space as well as in kernel space. The key concept was the re-randomization of code during runtime. Snow et al. ([125, 126]) stated that fine-grained ASLR could be circumvented by finding possible *ROP*-gadgets at runtime. Shioji et al. [127] presented code shredding, a technique to improve the randomization granularity (when compared to ASLR) to one byte. However, code shredding resulted in a tremendous overhead in both runtime and memory. Due to the cats and mouse game that defenders and attackers are playing, several new attack techniques against poorly implemented ASLR systems were presented throughout the last years: *Brute-Forcing* [128], *Partial Pointer Overwrites* [129], and *Return-Oriented Programming* [125].

Chen et al. [130] presented *CodeArmor* to harden code diversification against advanced *ROP* exploits. Their system improved on ASLR randomization and also introduced honey gadgets, which triggered an alarm in the case they were utilized.

The presented systems are different from *Bee Master*. They only focus on the exploitation stage. Modern malware does not need to exploit the system once it resides on it to inject code (see Chapter 4). This is due to the fact that many operating systems offer such capabilities via system calls or special APIs (e.g. Windows [54]). Therefore, *Bee Master* monitors its child processes for modifications to detect HBCIAs independently of the injection method and the underlying OS.

Instructions

Another branch of research focused on the randomization of the processor's instruction set. This branch of research has been actively followed at least since 2003. Even though the results have been promising and the performance of the approaches has increased throughout the years, the application to productive operating systems is still missing.

Cohen [131] argued that different obfuscated variants of a program raise the bar for new attacks significantly. His idea focused on the obfuscation and also implied randomization of the program code. He applied techniques like instruction reordering, random garbage insertion, and exchanging equivalent instruction sequences.

Barrantes et al. [116, 132] presented a *Randomized Instruction Set Emulator* called RISE, scrambling each byte of the legitimate program code. The foundation of this obfuscation was a pseudo random number generator (PRNG) that was seeded with a random key. When the key was known, the descrambling of the bytes was trivial. However, it was infeasible to produce even a short sequence of code with a meaningful behavior without knowledge of the correct key. Incorrectly scrambled code produced invalid instructions that crashed the program on execution. Similar to Barrantes et al., Kc et al. [24] proposed a system for thwarting code injection attacks by randomizing the instruction set of each process. It required special support by the processor. Several publications followed these two publications of Barrantes et al. and Kc et al. on instruction set randomization (e.g. [23, 101, 133–135]). They focused on alternate implementations, enhancements, or improvement of the performance.

Whereas all of the presented seminal papers prevent and potentially detect the execution of injected code, they require profound modifications of current operating systems to work. Such modifications are not realistic in the most popular closed source operating systems in the short term. *Bee Master* is different from these publications. First, it

does not rely on modifying current operating systems. It can be easily ported to other platforms without any modification of them. Second, it does not focus on prevention but rather on detection. The above mentioned papers focused mainly on prevention and detection may only be a side effect.

System Calls

Programs receive inputs and output results. This is achieved via the interaction with the OS kernel through system calls.

Chew et al. [136] presented the randomization of system call mappings. They changed these mappings in the kernel and dynamically rewrote system calls in programs before loading the program. Injected code did not know the correct mappings and hence crashed the victim process. Oyama et al. [137] proposed the encryption of system call IDs and arguments. Hence, an attacker without knowledge of the encryption algorithm and key was not able to use system calls. Building upon former approaches, Jiang et al. [138] introduced *RandSys*, an approach that combined ASLR with a weakened form of ISR (see previous section). They only randomized system calls. Similar to Jiang, Nguyen et al. [139] implemented a native API randomization for Microsoft Windows.

The presented publications can protect against code injections, especially in the classical case of remote exploits. However, they fail to protect against code injections carried out by an attacker that already resides on the system and utilizes legitimate system calls for foreign process space manipulation, e.g. due to access to the compiler tool chain. This is often the case with today's malware. Our approach *Bee Master* does not suffer from this limitation. It is able to detect such HBCIAs that are carried out by legitimate system calls, since they result in measurable changes in its child processes.

Operating System

Another set of proposals built diverse operating systems, which may increase the robustness against replicated attacks.

Pu et al. [140] discussed ideas for countering malware and exploits, which relied on implementation details of the OS. They counted on variation of the OS code. Even though the authors might be the first to apply biological ideas (e.g. the human immune system) to OSes to counteract malware and exploits, their publication contained vague statements and suggestions for future work. Forrest et al. [141] were the first to formally introduce the idea of building diverse operating systems. They argued that

diversity implies robustness in biology. Current computer systems were not very diverse and therefore very susceptible to mass exploitation attacks. They did not present a concrete solution to this problem but they proposed possible implementations like memory layout randomization and code permutation. Please note that these ideas were subject to research several times in the last two decades (see the previous sections). Chew et al. [136] proposed OS randomization to mitigate buffer overflow attacks. Since their proposal aimed at the mitigation of buffer overflow attacks, it randomized parts of the OS that played an important role in buffer overflow attacks: system call mappings, library entry points and stack placements. We have already discussed these techniques in the previous sections. Larsen et al. surveyed automated system diversity [142].

Our work is different from the presented publications. We aim for practical and applicable solutions such as *Bee Master* and *Quincy* that can be employed with current operating system implementations. Diverse operating systems pose many drawbacks. They may be problematic for system administrators and programmers when bugs occur. Also, they demand the reimplementation of current operating systems from the ground up.

3.2.3 Integrity

Next to randomization, integrity-based systems are popular in the literature. They ensure the integrity of the code and the code execution environment. First, we review work that focuses on the integrity of control and data flows. Second, we discuss further approaches that, for example, aimed at the integrity of the execution environment.

Control and Data Flows

Code injections violate the integrity of the control and data flows of the victim process. For instance, the hijacking of a thread violates the control flow of the corresponding victim process.

Kiriansky et al. [143] presented *Program Shepherding*, which verified every branch instruction and enforced certain security policies. For example, they detected code injections due to a code origin policy. Their system was based on the interpretation of binary code. Suh et al. [144] introduced *Dynamic Information Flow Tracking*. Their approach marked data coming from Input/Output (I/O) sources as spurious. Then, it tracked this marked data and detected dangerous uses of it. Abadi et al. [145, 146] defined the concept of *Control Flow Integrity* (*CFI*). Their idea ensured that the program execution followed a path in the precomputed program's control flow graph. Valid paths were determined beforehand. Younan et al. [93] proposed separating pointer and control

flow information of programs to mitigate a wide range of attacks including buffer overflows. They extended a C compiler to generate code with these characteristics. Castro et al. [147] proposed *Data Flow Integrity*. They statically computed a data flow graph and enforced valid data flows by instrumenting the program. Their system raised an exception when it detected an invalid data flow. Akritidis et al. [148] introduced a system that statically computed the control flow graph and determined memory access locations for each instruction. Then, code was generated that prevented the manipulation of objects by instructions that were not allowed to write to these objects. Bletsch et al. [149] presented *Control Flow Locking*. They lazily validated control flow changes after they occurred. Therefore, they inserted code before each indirect control flow transfer such as returns and indirect jumps. This code checked a memory value to verify if the current control flow had already been transfered. They aborted the execution in this case. Zhang et al. [150] improved *CFI* by introducing a springboard section, where they stored all legal targets of indirect control flow instructions. The order of the targets was randomized and access was restricted. The authors claimed that this technique worked directly on the binary code and significantly increased the performance of *CFI*. *CFI* continues to be an open field for research (e.g. [151–154]). Even though *CFI* raised the bar for attackers significantly, there are still ways to circumvent it, albeit certain conditions have to be met (e.g. [155–160]).

In contrast to the presented publications, *Bee Master* focuses on the detection of malware instead of exploit detection and prevention. We detect changes in the *Worker Bees* such as new threads and new libraries that are a result of executing malicious code. We assume that there are some attacks like simple *ROP* exploits that might be missed by *Bee Master* but there are other cases in which it should perform better than a *CFI*-based system. Therefore, a combination of it and *CFI*-based systems would be a way to boost detection rates even further and also beyond malware.

Other Approaches

Feng et al. [161] utilized call stack analysis to detect several attacks. They extracted call addresses from the call stack and tried to find a path with abstract execution between two program points. If shellcode was executed then a valid path did not exist. Linn et al. [162] ensured that only legitimate system calls were executed by adding information about the allowed system calls of a binary to a special section of the binary. Furthermore, they obfuscated system call traps using exceptions. Ratanaworabhan et al. [163] presented a system against *Heap-Spraying Attacks* that also targeted type-safe languages like Java. They monitored the entire heap and detected large-scale modifications to the heap's objects that are required in such an attack. Salamat et al. [164] proposed

multi-variant execution. They executed multiple variants of a program and compared their states at certain execution points. If there were discrepancies then an exception was raised. Philippaerts et al. [165, 166] presented *Code Pointer Masking*. They masked code pointers and therefore restricted the range of addresses that these pointers could point to. This should have ensured that an attacker could not make a pointer point to injected code. Baiardi et al. [18] introduced the separation of code execution and control flow in two virtual machines. With this clean separation, they were able to detect injected code. Chen et al. [167] implemented *Shreds* that allowed in-process private memory. They offered programmers a set of primitives that they had to include in their program. A modified compiler compiled the source code and a kernel driver ensured that only the owner of this in-process private memory accessed it. Their system countered data theft and manipulation by other threads of the process. Yun et al. [168] proposed a system to sanitize the utilization of API calls. It analyzed the source code with symbolic execution and determined semantically correct sequences of API calls. If a program utilized a different sequence at runtime then they detected this behavior.

Bee Master is clearly different from the presented publications. It neither requires a new OS or even a new computer architecture (e.g. [18, 163, 164]), nor is it tailored only to one platform (e.g. [162]), nor does it rely on the availability of source code (e.g. [165, 166, 168]).

3.3 Detection of Code Injection Attacks

In Section 3.2, we have already discussed the prevention of code injection attacks, which can also be leveraged for the detection of them in some cases and to a certain degree. This section discusses approaches that merely detect code injection attacks. First, we discuss the detection of *Host-Based Code Injection Attacks* (HBCIAs), the most related area to this thesis. Second, we survey *Remote Code Injection Attacks*.

3.3.1 Host-based Approaches

Even though there have been many publications on preventing code injection attacks (e.g. [23, 24, 120, 138, 145]), the research community has not intensively focused on detecting HBCIAs. This holds true especially in the case of malware. In the following, we discuss dynamic and static approaches separately. This corresponds to the two systems that we developed: Bee Master (dynamic) and Quincy (static).

Dynamic Approaches

Hanel [169] published a tool to detect HBCIAs in Windows processes. He achieved this by scanning each process for certain low-level characteristics. First, it checked if a library had been automatically injected into a process via certain registry keys (*AppInit_DLLs* and *APPCERTDLLs*). Second, the tool searched for private memory regions that might have been mapped into the process during a code injection. Third, it dumped all memory regions with *RWX* permissions. Furthermore, it could spawn an instance of the browser *Internet Explorer* and scanned it for these aforementioned characteristics. Hanel's implementation suffered from relying on low-level details, non-portability as well as not being extensible in order to detect a larger set of malware families. *Bee Master* abstracts from these low-level features. Therefore, it can also detect memory regions with other permissions like *RX* that have been injected into one of its *Worker Bees*. Hanel's tool focused more on forensic at runtime. Therefore, it is more comparable to *Quincy* than to *Bee Master*. *Quincy* employs the two last of Hanel's features. However, it does not rely on them but rather on a more comprehensive set of up to 36 features. On one side, Hanel's approach suffers similarly to other approaches like *Malfind* from false positives due to dumping all *RWX* memory regions. On the other side, it relies on information directly provided by the compromised system, which *Quincy* does not.

Xu et al. [170] employed taint tracking to detect a wide range of common attacks such as SQL injections, buffer overflows, and command injections. However, their idea was only applicable to C programs and demanded direct transformations of the source code. Sun et al. [20] proposed a system for detecting HBCIAs by hooking certain system calls associated with this behavior. The hooking was performed in kernel mode in order to ensure higher privileges than the malware. Since Sun et al. relied on certain system calls, they depended on low-level OS details. Furthermore, Sun et al.'s system was not capable of detecting unknown code injection attacks, because it only hooked system calls known to be related to prior code injection attacks. Buescher et al. [22] proposed a system that detected illegitimate manipulation of browser API, e.g. code hooks. It was based on the assumption that malware employs HBCIAs to manipulate these APIs. Srivastava et al. [21] detected HBCIAs by correlating host-based and network-based data. Amongst others, they monitored selected low-level system calls, e.g. *NtOpenProcess* and *NtCreateThread*. They utilized virtual machine introspection to gather behavioral data. Snow et al. [19] presented a new hypervisor-based OS called *ShellOS* that was capable of detecting *Remote* and *Host-Based Code Injection Attacks*. They achieved this by detecting shellcode in network streams and process buffers. They executed network streams and processed buffers at each offset. This was necessary since the offset to possible shellcode was unknown. Their system employed various detection heuristics based on OS-specific

structures like the *Process Environment Block* (PEB). Strackx et al. [171] proposed an approach for hardening stock operating systems against kernel-level and process-level malware. It consisted of a compiler and a runtime security architecture. The compiler compiled programs in a way so as to they had a public and a secret section. Whereas the public section was accessible by everyone on the system, the secret section was only accessible by the module itself. Therefore, the impact of kernel-level and process-level malware was significantly reduced. Korczynski et al. [172] presented *Tartarus*, a system to detect code injections and code-reuse attacks. It utilized dynamic taint tracking over the whole operating system to detect them. It did not rely on API call hooking. However, their system had a severe impact on the system performance and it was based on a virtualization software. Hence, it does not directly apply to bare-metal. Furthermore, they only evaluated their system on Windows XP SP3, which has a much smaller code footprint than Windows 7, 8, and 10. Thus, their approach might be infeasible on these newer platforms. *Bee Master* differs from the above publications: It neither relies on the source code of the victim programs (Xu et al. [170]) nor on the hooking of low-level system calls (Sun et al. [20], Buescher et al. [22], Srivastava et al. [21]) nor on fundamental changes to the operating system (Snow et al. [19], Strackx et al. [171]) nor on special virtualization platforms (Korczynski et al. [172]).

The anomaly detection of process behavior is also closely related to *Bee Master*. Its goal is to distinguish between normal and abnormal behavior of processes. Forrest et al. [173] proposed a method to detect anomalies in Unix processes. They recorded sequences of system calls and leveraged them to build process specific signatures beforehand. Then, they applied these signatures online to detect anomalies in the system. Warrender et al. [174] presented further data models for anomaly detection based on system calls. Sekar et al. [175] also based their work on Forrest et al. [173]. They presented a method to automatically and efficiently generate finite-state automatons to detect anomalous sequences of system calls. Wagner et al. [176] proposed the detection of anomalies in program behavior by applying static analysis to each program that runs on a system. Thereby, they modeled a transition system that detected anomalies in system call traces. Yap et al. [177] raised the bar for evasion attacks by introducing argument abstraction of system calls. Mutz et al. [178] took the system call context into account. This context comprised the return addresses placed on the call stack. Throughout the years many publications have evaluated diverse models for system call-based anomaly detection (e.g. *Hidden Markov Model* [179], *AdaBoost* [180], or *Neural-Trees* [181]) to decrease false positives. Even though these works are more general than *Bee Master*, they fail to detect an attack if the malware mimics the original application. *Bee Master* is not vulnerable to mimicry attacks since it does not depend on system call tracing. In addition, *Bee Master* leverages HBCIA-specific features including the number of threads

and loaded libraries. Furthermore, *Bee Master* does not hook system calls, which is noisy and easily detectable. Also, there is malware that actively tries to remove hooks (e.g. the dropper/rootkit *Andromeda*).

Static Approaches

The *Volatility* plugin *Malfind* proposed by Hale Ligh [25] detected malware in memory dumps similar to our method. *Malfind* implemented a combination of features designed to decide whether a memory region is benign or malicious. At first, it marked entirely empty memory regions as benign. Pages with *RWX* protections and unlinked libraries (from the *PEB*) were marked as malicious. Furthermore, *Malfind* detected wiped PE headers in *RWX*-protected memory regions. Unflagged regions were labeled as benign. Lassalle proposed *Malfinddeep* [25], an improvement to *Malfind* that utilized whitelisting of memory regions based on *ssdeep* hashes. We did not evaluate *Malfinddeep* since there was no official whitelist of hashes available. Our work significantly improves upon *Malfind*, which we showed in *Quincy*'s evaluation. It considers a superset of *Malfind*'s features and adds many more to improve detection performance.

Pek et al. [28] presented *Membrane*, a system to detect HBCIAs in memory dumps. *Membrane* reconstructed low-level memory paging information of Windows's software memory management unit (MMU) and leveraged this information to detect HBCIAs. They identified circa 25 features based on domain knowledge and utilized the machine learning algorithm *Random Forest* to detect HBCIAs on process-granularity. There are overlaps between *Quincy* and *Membrane* such as the implementation as a *Volatility* plugin and the utilization of one common feature ($memory_{mapped}$). However, *Quincy* significantly differs from *Membrane*. First, *Quincy*'s detection has a finer granularity. Whereas *Membrane* detects HBCIAs on a process-granularity, *Quincy* detects them on memory region-granularity. Therefore, a direct comparison between *Quincy* and *Membrane* is not possible. Second, the approach is very prone to noise. Their results were very good on Windows XP but on Windows 7 they showed 25% less accuracy. We assume that on Windows 10 this problem would be even worse since the noise level increases with every Windows version (see Table 7.5). Third, they implemented their approach to analyze two older Windows versions (XP and 7). *Quincy* also works on the latest version. Fourth, *Membrane* was based on low-level features. The authors had to reverse engineer parts of the Windows kernel to implement their system. Porting *Membrane* to a new Windows version or even new OS would require many hours of tedious reverse engineering.

Monnnappa proposed the *Volatility* plugin *Hollowfind* [27]. It detected a subset of all HBCIAs: *Process Hollowing* (see Section 4.3). The detection heuristic of *Hollowfind* consisted of two features that were engineered based on several *Process Hollowing* case studies. First, *Hollowfind* scrutinized the parent child process relationship, e.g. the process *lssa.exe* should not have been started by the process *explorer.exe*, which suggests *Process Hollowing*. Second, it revealed discrepancies in two process internal data structures (VAD tree and PEB) that are a tell-tale sign of *Process Hollowing*.

White et al. [26] described a system to detect the provenance of malicious code in memory dumps of Microsoft Windows. They achieved this by hashing memory pages and comparing these hashes to a previously built hash database. Thereby, they could reduce the amount of memory pages that have to be manually analyzed.

Quincy builds on and enhances current state-of-the-art approaches like White et al. [26], Pek et al. [28], and *Malfind* [25]. First, it considers further features to detect HBCIAs such as *Thread Entry Points* and *High Entropy Areas*. Second, we proved in Chapter 7 that *Quincy* outperformed *Malfind* and *Hollowfind*. It showed up to 63% less false positives and up to 27% more true positives when compared to *Malfind*. This general improvement yielded an increase of the AUC score of up to almost 9% (see Chapter 7).

3.3.2 Network-based Approaches

A good overview of network-based code injection attacks was provided by Polychronakis et al. [95] and, more recently by Fritz et al. [182]. Their studies discussed the basics of network-based code injection attacks and pointed out current trends. Pure network-based systems were restricted to payload detection in the network stream (e.g. [183–187]).

The presented work is different from *Bee Master* and *Quincy*. It focused on the detection of malware on the network layer. Whereas this may seem favorably at first, these works have many drawbacks in the context of malware detection. First, tricking the victim via social engineering into downloading the malware is one of the preferred ways to spread malware [188]. If this is combined with reasonably good obfuscated network traffic, then this renders a pure network-based detection unlikely. Second, there are many more ways malware can infect a system, including USB drives and air gaps [189]. In the end the malicious payload is executed on the host, where we detect its actions.

We provided an overview of honeypots in Section 2.3. Honeypots detect illegal access to computer systems by providing attackable resources similar to *Bee Master*. They have been intensively studied in the literature during the last years. The majority of honeypot

research focused on network attacks. This includes honeypots that are waiting to be exploited (server honeypots [59, 79, 190]) and honeypots that are actively trying to be exploited (client honeypots [191–193]). *Bee Master* does not apply to network-based attacks but rather to attacks on local processes. Hence, current (network-based) honeypot systems differ in their focus from our work.

3.4 Non-Scientific Work

Malware is a menace to society (see Sections 1 and 2). This means that there is a high demand for protection, which in turn creates a huge market for computer security products. For instance, *Cybersecurity Ventures* predicted that the global computer security spending will be more than one trillion US dollars, just in the period from 2017 to 2021 [194]. Therefore, many private companies are investing in computer security research with the objective of gaining high revenues.

Since the field of malware analysis is a very practical one and the computer security industry also significantly invests into applied research, we survey patents and products related to code injections in the following.

3.4.1 Patents Related to Code Injections

Several patents related to code injections have been filed throughout the last years. Even though patents are not classical scientific work published in peer-reviewed proceedings or journals, they can be related, because they may be based on scientific work and results.

Krishnan et al. [195] hold a patent for injecting code into existing application code. At first, they statically manipulated an existing application. They either manipulated the import table or added a loader stub for loading another library. Once the application had been executed, the library was loaded and the injected code executed. Ghizzoni [196] holds a patent for injecting code into another process. Heasman et al. [197] filed a patent for a system that stored instructions transformed, fetched and retransformed them, and finally executed them, foiling code injection attacks. Their patent seems to be similar to ISR-based approaches. Jin et al. [198] hold a patent to detect code injection attacks based on memory page updates. Mensch et al. [199] filed a patent to inject code at runtime to perform checks, e.g. to restrict the execution of an application to a certain platform. Yun [200] has a patent to monitor code injections. His patent is based on API call hooking, which is commonly associated with code injection attacks

(e.g. *WriteProcessMemory*). Park's patent [201] prevented code injections, similar to the patent of Yun. But his idea was based on hooking system calls in the kernel.

All these patents differ from our work. Some of these patents merely describe code injection attack techniques. Some also describe prevention techniques. These techniques are either based on fragile hooking (e.g. [200, 201]) or demand new computing architectures (e.g. [197]). We have already discussed in the previous sections why these two assumptions are not desirable.

3.4.2 Security Products Related to Code Injections

Recently, many endpoint security products include machine learning techniques such as *CrowdStrike Falcon* [202], *Cylance CylancePROTECT* [203], and *Barkly Endpoint Protection Platform* [204]. Unfortunately, these companies just advertise the use of machine learning but they are vague about the details, for instance, the features, the machine learning algorithms, and if they detect code injections in particular.

Lately, Microsoft has added support for detection of HBCIAs to their commercial product *Windows Defender ATP* [205]. In a series of blog posts, they outlined the internals of their product [206–208]. It focuses only on the last Windows 10 and runs as an operating system component, i.e. in kernel space. In a nutshell, it is a dynamic behavioral anomaly detection approach. It instruments API calls like *VirtualAllocEx* and *QueueUserAPC* to detect abnormal behavior. It applies machine learning-based models to detect anomalies due to code injection techniques including *Process Hollowing* and *Atom Bombing*. *Windows Defender ATP* is in line with many similar approaches such as Sun et al. [20], Warrender et al. [174], and Hu et al. [180]. However, *Windows Defender ATP* is different from our proposals. First, it naturally differs from *Quincy* since it is not a forensic approach. Second, it resembles aforementioned systems that exhibit properties like API call hooking/instrumentation and operating system dependency. We have already discussed why these properties are undesirable.

In general, it is difficult to tell what these products exactly do without knowledge of their internals. Since their source code is a trade secret, the only way would be to tediously reverse engineer them. However, this would require on one hand many hours if not days of hard work and on the other hand it may not be legal. Hence, there is no way to achieve a fair comparison between these products and our approaches *Bee Master* and *Quincy*.

3.5 Conclusion

This chapter has surveyed work of several closely related fields. It has shown that current approaches are different from our work. We are the first to formalize HBCIAs in the context of malware. Furthermore, we propose two methods to detect this attack dynamically as well as statically. Our first approach *Bee Master* dynamically detects HBCIAs by employing fake processes as honeypots. We are the first to transfer the honeypot paradigm to operating system processes to detect HBCIAs. *Bee Master* is platform-agnostic as well as operating system-agnostic. It differs significantly from related work: It neither relies on fundamental changes to the OS or even the hardware (e.g. [19, 23, 116]), nor on low-level API hooking (e.g. [21, 22]), nor on the availability of the victim's source code (e.g. [171]). Our second system *Quincy* statically detects HBCIAs in memory dumps by employing machine learning techniques. It builds on the state of the art *Malfind*, which it significantly enhances and hence it improves on its performance. It differs from other related publications such as Pek et al. [28] due to a finer detection granularity and White et al. [26] due to a more generic idea that does not rely on whitelisting. We will come back to some of the surveyed publications later, compare them to the requirements to detect HBCIAs, and show that current systems do not fulfill these desirable requirements.

Die Wissenschaft fängt eigentlich erst da an, interessant zu werden, wo sie aufhört.

Justus Freiherr von Liebig

4 Defining Host-Based Code Injection Attacks

We face a steadily increasing amount of malware samples today [3, 4, 209]. Cyber criminals utilize malware for a multitude of activities, e.g. credit card fraud [33] or supposedly use it for espionage [6] (see also Section 2.1). Due to the diversification of operating systems, malware targets not only Windows but also, amongst others, macOS, Linux, and Android (e.g. [10, 11, 210]).

We see an increase in malware and target operating systems. Similarly, malware authors continuously develop new techniques to hide their malware. A popular technique amongst malware is the *Host-Based Code Injection Attack* (HBCIA). HBCIAs enable malware to execute its code within the context of a benign process. This contradicts the idea that only one program is accountable for the behavior of a process. Throughout the last years, many reports have uncovered malware families that secret services allegedly implemented and distributed. These families operated for a very long time before they were discovered (e.g. *Careto* [13], *Stuxnet* [12], or *Uroburos* [211]). Although their developers had different ideas in mind, they share a common feature. They all employ local code injections into benign processes.

Covert operation is one reason to conduct HBCIAs. But local code injections are not limited to targeted malware and covert operation is not the only reason to employ them. For instance, crimeware families such as *Citadel* [212], *Conficker* [46], *Dexter* [213],

and *ZeroAccess* [214] utilized them to access critical information, to covertly operate, to escalate privileges, or to manipulate security products.

The malware families that we have referred to in the last paragraph focus on Windows. Yet, HBCIAs are a platform-independent problem. Widespread operating systems are all vulnerable to this attack. Malware such as *Flashback* (macOS) [11], *Hanthie* (Linux) [10], and *Oldboot* (Android) [210] showed this on mobile and non-mobile systems.

This chapter defines the phenomenon of *Host-Based Code Injection Attacks* in the context of malware. We build a solid foundation for future research on this topic by defining them. This includes a clean separation from the far more known remote code injection attacks that computer worms commonly employ. We discuss the advantages and disadvantages of this attack from a malware author's point of view. This gives the reader a first impression of why this attack is popular among current malware. To study HBCIAs in detail, we present the algorithms behind HBCIAs. In practice, we can observe a variety of HBCIA models. After this chapter, the reader should have a general understanding of the theory. The next chapter will pick up this theory and show in several measurements the situation of HBCIAs in the real world.

The remainder of this chapter is structured as follows: The next section defines code injections in general and HBCIAs in particular. Then, we discuss motivations of malware authors in utilizing them. Thereafter, we scrutinize HBCIA algorithms. In the course of this section, we elaborate a formal model of these algorithms and a taxonomy to classify them. Finally, we conclude this chapter with a discussion of possible future developments regarding HBCIA-employing malware and a conclusion of this chapter.

Please note that partial results of the presented chapter were published in the seminal paper "Host-Based Code Injection Attacks: A Popular Technique Used by Malware" at the conference "Malicious and Unwanted Software: The Americas" (MALWARE) in 2014 [15]. The proceedings were published by *Springer International Publishing*.

4.1 Defining Code Injections

The first code-injecting malware was the Morris worm in 1988 [215]. It infected large parts of the Internet via remote exploitation of a buffer overflow. Several code injecting computer worms followed thereafter. All of them utilized *Remote Code Injection Attacks* to spread. However, the consequent deployment off-the-shelf routers with restrictive default firewall settings in the early 2000s ended the era of huge worm outbreaks [216]. Malware families like *Zeus* began to employ *Host-Based Code Injection Attacks* in the

middle of the 2000s [33]. They enabled, for instance, the malware to intercept banking information from within browsers.

In this section, we discuss *Code Injections* and more specifically *Host-Based Code Injection Attacks* as employed by malware. At first, we define the attacker model that we adhere to during the rest of this thesis. Then, we define the term *Code Injection* and the terms *Host-Based* and *Remote Code Injections*. Afterwards, we differentiate between *Host-Based/Remote Code Injections* and *Host-Based/Remote Code Injection Attacks*. Whereas *Remote Code Injection Attacks* have been intensively researched (e.g. [185, 217]), there is less profound research on *Host-Based Code Injection Attacks* (see Chapter 3).

4.1.1 Attacker Model

This section introduces the attacker model that the rest of this thesis follows. Since we deal with a local attack, there are not many strict assumption that we make. Most importantly, we assume that the malware binary already resides on the target system. Otherwise, we refer to a remote code injection. When launched, this malware targets another process on the same system and tries to inject code into its process space. We do not make an assumption about the way the malware binary was transfered to the system. There are several ways to achieve this, including drive-by-downloads [39], a download by the user due to social engineering [218], and infected removable mediums [76]. Likewise, we do not make any assumption about by whom and how the malware is executed. This can be, for example, due to social engineering or automatic shellcode execution. Furthermore, it does not matter what privilege level the malware has during execution. However, the privilege level influences a successful HBCIA.

4.1.2 Code Injections

We begin by defining simple *Code Injections* (CI) in Definition 4.1 as we defined them in [15].

Definition 4.1. "A *Code Injection* denominates copying of code from an injecting entity ϵ_{inject} into a victim entity ϵ_{victim} and executing this code within the scope of ϵ_{victim} without the victim's knowledge." [15]

ϵ_{inject} and ϵ_{victim} may be, for instance, hardware devices that run firmware and can manipulate the main memory or user space processes. Note that there are two things crucial to a *Code Injection*. They are the (injected) code and the execution context of this code. We show in Section 4.3 that code injection algorithms always reflect them.

4.1.3 Host-Based and Remote Code Injections

Definition 4.1 does not specify where ϵ_{inject} and ϵ_{victim} reside during a CI. We differentiate between two cases: both reside on the same system (*Host-Based Code Injection*) or both reside on different systems (*Remote Code Injection*). In the latter case, they have to communicate via a communication channel, e.g. a computer network. We define a *Host-Based Code Injection* (HBCI) in Definition 4.2 as we defined it in [15].

Definition 4.2. "A *Host-Based Code Injection* (HBCI) is a Code Injection where the two entities ϵ_{inject} and ϵ_{victim} reside on the same computer system." [15]

ϵ_{inject} typically injects code into ϵ_{victim} by querying the OS to do so via APIs/system calls. Hence, ϵ_{inject} would be a user space process. However, it is also possible that ϵ_{inject} resides in kernel space as a kernel module or in the firmware of a hardware device. In these scenarios, the implementation of a HBCI would involve more work than just calling a couple of APIs.

This thesis focuses on injections that occur on the same host system. For the sake of completeness, we define *Remote Code Injections* as well. In contrast to HBCIs, ϵ_{inject} and ϵ_{victim} reside on two different systems during a *Remote Code Injection* (RCI). We define RCIs as we defined them in [15].

Definition 4.3. "A *Remote Code Injection* (RCI) is a Code Injection where the two entities ϵ_{inject} and ϵ_{victim} do not reside on the same system. They interact by means of a communication channel." [15]

A communication channel, amongst others, is a computer network. For instance, computer worms utilize this channel to attack their victims [219]. They exploit a vulnerability in a network service. ϵ_{inject} is a network client and ϵ_{victim} a network service during an RCI. ϵ_{inject} sends a malicious payload to ϵ_{victim}. If ϵ_{victim} is vulnerable, it executes the contained exploit code. For example, the computer worm *Conficker* exploits a vulnerability in the *SMB* network service in order to hop from end point to end point [46].

4.1.4 HBCI/RCI vs. HBCIA/RCIA

CIs are not malicious per definition. We identify various legit use cases for CIs. These include hot patching [220], software diagnostics [196], malware analysis [221], and debugging [222]. Furthermore, Microsoft holds a patent regarding code injections titled "Method for injecting code into another process" that states legit use cases: "[the] invention relates generally to computer software diagnostic tools" [196]. However, the aforementioned use cases do not apply to normal applications like browsers, office suites,

and encryption software. They do not require code injections to implement their functionality. All legit use cases that we have stated above relate to application development and malware analysis, which strengthens our suggestion. We can not differentiate between HBCIs/RCIs, if we do not take their purpose into account. We define a *Host-Based Code Injection Attack* (HBCIA) and a *Remote Code Injection Attack* (RCIA) similar as we defined them in [15]:

Definition 4.4. If a *Host-based Code Injection* or a *Remote Code Injection* serves a nefarious purpose, then it is called a *Host-Based Code Injection Attack Remote Code Injection Attack*, respectively [15].

Nefarious purposes are, amongst others, scraping of information, extraction of cryptographic keys from the process space, or patching licensing algorithms in memory. We assume throughout this thesis that due to the purpose of malware in general, all its code injections are malicious. Hence, it is an attack. Definition 4.4 is subjective. A malware author may assume the application of code injections to analyze their malware as an attack.

In the following sections, we only focus on HBCIAs. Whereas RCIAs used to be very widespread in malware at the beginning of the 2000s and consequently they were thoroughly investigated (e.g. [185, 217] but also see Chapter 3), HBCIAs became popular around ten years ago (e.g. [20, 21, 24, 169]).

4.2 Advantages and Disadvantages of HBCIAs

We have defined HBCIAs in the last section. In particular, we have distinguished them from remote code injections. So far, we have shunned their benefits. Or in other words: Why are they so popular with modern malware? Malware families such as *Citadel* [212], *Flame* [223], and *Flashback* [11] utilized HBCIAs to successfully operate on various operating systems.

This section considers HBCIAs from a malware author's perspective. In doing so, we show their advantages such as privilege escalation and covert operation. This may explain their current popularity with malware authors (see our problem size estimation in Section 5.1). But this comes with a price tag. Malware authors have to meet additional challenges when implementing code-injecting malware. This includes, for instance, system stability and increased architectural complexity.

4.2.1 Advantages of Employing HBCIAs

Leveraging HBCIAs comes with many benefits for malware authors. They allow malware to intercept critical information, to escalate privileges, to covertly operate, and to manipulate security products. We discuss each of these benefits in the following sections.

Interception of Critical Information

Computers process personal and critical information such as medical data, credit card information, or even classified documents. This information is valuable. To some this means money, to some this means an increase in (political) power. Therefore, cyber criminals and other threat actors wish to access this information. Since this information is usually transfered between computer systems in an encrypted form, they need to access it on a host system. In many cases this data never touches the hard disk in an unencrypted form. It just stays unencrypted in memory for a short period of time. As a consequence, this information needs to be intercepted when it is processed in clear text in memory. Hence, the corresponding process spaces are targeted by malware in order to intercept this information.

After a successful injection into ϵ_{victim}, the injected code can access all information that ϵ_{victim} handles. For instance, on Windows and Linux, there is no access restriction within a virtual process space. Hence, the injected code may read and write code too much as data as it wishes. As a consequence, malware may intercept critical information in clear text just before it is encrypted and sent over the network, for example, in a browser. To achieve this, the malware may hook cryptographic APIs of browsers. This is also known as a *Man-in-the-Browser* attack and typically banking Trojans like *Citadel* [212] operate this way. Another scenario is to scrape the process space to discover valuable data like credit card numbers. This behavior exhibits Point-of-Sale (PoS) malware like *Dexter* [213]. Note that there are academic approaches like *Shreds* [167] that implement private memory within a process space. However, major operating systems like Windows and Linux have not implemented such techniques so far.

HBCIAs are utterly important to banking Trojans since they enable them to intercept and manipulate banking sessions in clear text. The only way for them to intercept this data is in user space because the browsers utilize user space libraries to encrypt it [22]. Clear text interception in kernel space is, for this reason, not possible. Additionally, current cryptographic algorithms like RSA or AES are still not feasible to attack in practice (given a reasonable key length).

Example: The banking Trojan *Citadel* injects code to intercept critical information like banking credentials [212]. It injects its code into all accessible processes that have not been infected yet. If *Citadel* notices that it resides in a browser, it hooks the browser's encryption functions to intercept and manipulate unencrypted network traffic sent to the victim's banking institutions.

Privilege Escalation

On modern operating systems it is necessary to hold certain privileges in order to conduct certain actions. Typically, these privileges are granted to a user. For instance, on Linux there are regular users that hold some higher privileges to work but lack some privileges, e.g. to directly write to the hard disk's boot sector. However, there is the superuser who is privileged and who could conduct the aforementioned action.

Malware utilizes code injections to escalate privileges. The injected code is executed with the same privileges as the target process. There may be a discrepancy between the privileges of the attacker and the victim process. For example, sometimes processes are allowed to inject code but are not allowed to communicate via the network or to access protected files. A solution to this problem is to inject code into another process with sufficient privileges and conduct the operation from there.

Example: The PoS malware *Dexter* attacks *Internet Explorer* with an HBCIA to circumvent local personal firewall policies [213]. A reason for this behavior is that *Internet Explorer* is whitelisted by these firewalls. Therefore, *Dexter* can communicate through this browser with its server without being blocked and without raising suspicion.

Covert Operation

The next reason to employ HBCIAs is covert operation. Malware strains like espionage toolkits and banking Trojans may be required to operate for a longer period of time because the information they are after is processed/accumulated over time. Therefore, it is rather common that a malware operates for weeks, months, or even years (e.g. [224]) on a target system rather than a couple of hours.

Malware that requires its own process space may be suspicious. It may utilize a fishy name, it may duplicate a process name of a system process like *explorer.exe*. However, the user may detect this fishy process and start its own investigation. A consequence could be that they kill the process and stop the operation of the malware. Therefore, malware uses HBCIAs to overcome this issue. Since its code runs in another process, it

blends into the behavior of it. Hence, no own process is required and the malware may continue its operation for a longer period of time. This may evade security products, since the malware's operation blends into the behavior of ϵ_{victim}, e.g. determined by the system call sequence.

Example: The cyber espionage malware *Flame* carries out HBCIAs to covertly operate [223]. It accomplishes this with the help of an unusually complex chain of injections through several system processes on the target system. The infected processes execute different payloads and therefore they are responsible for distinct features of this malware. This effectively increases the difficulty of detection because the malicious behavior is split over distinct processes.

Manipulation of Security Products

In the previous section about covert operation, we have stated that many families aim to operate for a longer period of time. They wish to do so without being disturbed. Security products like antivirus software jeopardize this objective. Hence, the manipulation of security products is a key to long term operation.

Security products like antivirus software are far from perfect. They contain bugs as any other sofware system and therefore increase the attack surface [225]. Since a security product's goal is detection and removal of malware, malware may act in self-defense. This means that it tries to provoke a denial of service of the security product or it tries to completely remove it with the objective to continue its operation without interruption. HBCIAs are a way to achieve this.

ϵ_{inject} may inject code into security products. After successful injection, there are a plethora of manipulation options. The first option is to terminate ϵ_{victim} from within. Observing processes assume that this termination was intended by ϵ_{victim}. A second option is code manipulation, e.g. hooking functions or overwriting essential code sections with no operation instructions.

Example: The rootkit *ZeroAccess* [226] monitors the access to a mock-up executable that poses as a honeypot to antivirus scanners. It terminates every process that accesses this file by injecting shellcode into it.

4.2.2 Disadvantages of Employing HBCIAs

Section 4.2.1 has shown the advantages of HBCIAs from a malware author's point of view. But there are also disadvantages compared to malware that resides in its own

isolated process space. These disadvantages include increased architectural complexity and the risk of system instability.

Architectural Complexity

The first disadvantage of HBCIAs is the increased complexity of the malware. This implies higher development costs owed to the fact that the malware authors are required to implement a parasitic component for other processes instead of a self-contained, single process malware.

Some malware families resemble complex distributed systems. We define the nodes of such a distributed system as the infected processes that communicate with each others to achieve a common goal. Hence, the malware author faces that same challenges that developers of distributed systems face. These challenges include, for instance, timing, failure tolerance, message passing, and synchronization [227]. In addition, testing and debugging of distributed systems is more difficult than in the case of a single process.

Example: The banking Trojan *Zeus* injects its code into several processes. Not all processes serve for equal tasks. *Zeus* abuses the *Winlogon* process as master process. The other infected processes are daemons of this master process. Messages are passed via a named pipe between the master process and its slaves [33]. Notably, the bot itself logs debug messages to a debug server via a named pipe when in debug mode (see original source code of *Zeus* [38]). The developer of *Zeus* added this feature to facilitate the development.

System Instability

As every software, malware may contain bugs. Each bug is a risk and its occurrence may terminate the malware. Therefore, injecting code into ϵ_{victim} poses a risk since the bugs of the malware also influence the victim process. If the victim process is a critical system process then this may lead to system instability. As a consequence, the user may start investigations and detect the malware, which would lead to the removal of it. Therefore, it is important that malware authors consider stability, for example, a test-driven development approach to increase stability and ensure a close to bug free software could be a solution to this problem. The challenge that malware authors face is that they do not only need to think about their own code but also to think about the code of ϵ_{victim}. Above all, they have to ensure that there are no interferences between the two code bases like race conditions.

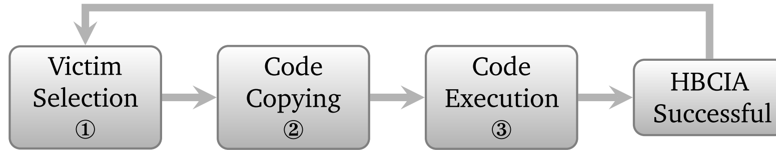


FIGURE 4.1: Simplified HBCIA algorithm comprising three steps. First, the algorithm selects a victim. Second, it copies the code into the victim. Third, it triggers the code execution within the victim. Possible failure states are omitted.

Example: The espionage malware *Careto* poses a sophisticated structure with regards to these issues [13]. For instance, *Careto* monitors the process termination of its host process to free unused handles. This may be one of the reasons why this malware operated almost seven years without attracting any attention.

4.3 HBCIA Algorithms

We have motivated why HBCIAs are a popular malware feature in the last section. But we have also seen that there are additional challenges when using them. This section examines HBCIA algorithms. They are the foundation of every HBCIA. In practice, we can observe different models that malware employs. For instance, banking Trojans like *Zeus* [33] inject their code into as many victim processes as possible in order to execute its code in parallel to the victim's code. This allows them to intercept and manipulate information that the victim's code processes. Understanding the HBCIA algorithm models presented in the following should allow readers to understand any HBCIA algorithm found in the wild.

In the following sections, we assume that ϵ_{victim} is a process, since today's malware usually targets user space processes. The attacker entity ϵ_{inject} may be a process (e.g. [36, 38, 228]), a kernel module (e.g. [214]), or even a hardware device. Figure 4.1 outlines the basic idea of an HBCIA algorithm. ① At first, ϵ_{inject} selects a victim process ϵ_{victim}. ② Then, ϵ_{inject} copies code into it. ③ Finally, ϵ_{inject} executes the injected code within ϵ_{victim}. This algorithm may repeat to inject code into other processes. Note that the steps can result in a failure state. However, we omit possible failure states in this figure for the sake of clarity. The following sections examine each of these steps. The last section closes with the proposal of an HBCIA algorithm taxonomy.

4.3.1 Victim Process Selection Strategy

Different processes process different data. Depending on the malware's intention, it has to carry out a selection of its victims. Hence, the attacker entity ϵ_{inject} selects a victim

entity ϵ_{victim} before the injection. There are two selection strategies: *Shotgun Injections* and *Targeted Injections*. We explain these two terms below.

Basic Definitions

Before we define *Shotgun* and *Targeted Injections*, we present the set of all possible ϵ_{victim} entities and the set of all accessible ϵ_{victim} entities from an attacker's point of view. This is necessary in order to precisely describe the set of victims in the models defined later.

We define $\mathbb{A}$ as the set of all running victim entities, whose formal definition is given in Definition 4.5 similar to the one we defined in [15].

Definition 4.5. Let $\mathbb{A} = \{\epsilon_{inject}, \epsilon_{victim_1}, ..., \epsilon_{victim_n}\}$ be the set of all running victim entities that an arbitrary ϵ_{inject} can see where $n \in \mathbb{N}_{>0}$.

Please note that $\mathbb{A}$ explicitly contains ϵ_{inject} since our model also assumes self-injection. For instance, a packer could leverage self-injection due to obfuscation reasons or a programming error in ϵ_{inject} may lead to self-injection. There may be many possible victim entities running on a system. However, they are not all accessible to the attacker. For instance, on Linux the command line tool *ps* also includes processes that the administrator has started. Yet, the malware may execute with lower privileges and hence it may not access the administrator's processes. As a consequence, we require a second set $\mathbb{B}$. This is the set of all victim entities accessible to ϵ_{inject} that it may inject code into. We define in Definition 4.6 similar to the one we defined in [15].

Definition 4.6. Let $\mathbb{B} = \{\epsilon_{inject}, \epsilon_{victim_1}, ..., \epsilon_{victim_m}\}$ be the set of all entities that are accessible to ϵ_{inject} and into which it can inject code where $m \in \mathbb{N}_{>0}, m \leq n, \mathbb{B} \subseteq \mathbb{A}$.

$\mathbb{B}$ is a subset of $\mathbb{A}$. In the special case where the user is the administrator both sets are equal, i.e. $\mathbb{A} = \mathbb{B}$. Please note that $\mathbb{B}$ can not be empty since we assume that ϵ_{inject} can at least carry out a self-injection.

Given the sets of victim entities, we can define two selection strategies that draw victims from $\mathbb{B}$. The first one is to exhaustively draw entities from $\mathbb{B}$ (*Shotgun Injections*). The second one is to selectively draw victims from $\mathbb{B}$ (*Targeted Injections*).

Shotgun Injections

The first victim process selection strategy is called *Shotgun Injections*. It denotes the act of blindly injecting code into all accessible victims. These victims may also include

important system processes. Definition 4.7 defines a *Shotgun Injection* similar to the one we defined in [15].

Definition 4.7. If ϵ_{inject} injects code into every ϵ_{victim} of $\mathbb{B} \setminus \{\epsilon_{inject}\}$ then this is denoted as *Shotgun Injection*. If it injects code into every member of $\mathbb{B}$ including itself then this is denoted as *Full Shotgun Injection*.

Please note that the injecting entity ϵ_{inject} is not requiered to inject into every member of $\mathbb{B}$ but only into every ϵ_{victim}, i.e. $\mathbb{B} \setminus \{\epsilon_{inject}\}$. Recall that $\mathbb{B}$ consists of a union of the set of all victim entities and ϵ_{inject}. However, there may be *Shotgun Injections* that inject code into every member of $\mathbb{B}$ including a self-injection into ϵ_{inject}. They are denoted *Full Shotgun Injections*. Though technically possible, we consider this a bug in the HBCIA algorithm.

The *Shotgun Injection* is a greedy victim process selection strategy. Its objective is to inject code into every running process on a system. As a consequence, this may lead to complications because the malware author can not anticipate the running processes on a particular system (see also system instability in Section 4.2.2). For instance, unintentional targets could be antivirus processes, which would raise suspicion.

Examples: Banking Trojans like *Zeus* [38] commonly adopt *Shotgun Injections*. They inject code into as many victim processes as possible to accomplish their goal of stealing credentials.

Targeted Injections

In contrast to shotgun injecting malware, malware that conducts *Targeted Injections* carefully selects its victims. It attacks a subset of all possible victim entities. We define a Targeted Injection in Definition 4.8.

Definition 4.8. Let $\mathbb{C} = \{\epsilon_{victim_1}, ..., \epsilon_{victim_o}\}$, where $o \in \mathbb{N}_{>0}, o \leq m$. If ϵ_{inject} injects code only into a subset $\mathbb{C} \subset \mathbb{B}$ or $\mathbb{C} \cup \{\epsilon_{inject}\} \subset \mathbb{B}$ then this is a Targeted Injection.

Malware that leverages *Targeted Injections* carries out a selection process. It detects its victim processes through one or several features. The process name is a commonly employed feature (e.g. Tinba [36]). For this purpose, the malware compares the process name of each running process to an internal list. If a process name matches then the malware attacks the corresponding process. There are more ways how malware can detect its victim processes. These may include signatures, the parent process, or open file handles.

An advantage of *Targeted Injections* is that they are less suspicious compared to *Shotgun Injections*. The malware can shun risky victims like system processes. Furthermore, malware does not unnecessarily need to waste system resources since it does not infect every accessible victim.

Example: Cyber espionage malware such as *Flame* [223] and *Stuxnet* [7] leverage *Targeted Injections*. It is crucial for espionage malware to carefully select its victims in order not to raise suspicion owed to, for instance, frequent system crashes or waste of system resources.

4.3.2 Code Copying

After having chosen a ϵ_{victim}, ϵ_{inject} copies its code into it. There are several ways how this can be accomplished. Examples are debugging APIs provided by the OS [196], direct memory access from kernel space [226], or a buffer overflow exploitation [7].

Example: Sometimes the copying of code is carried out in multiple stages. The computer worm *Conficker* copies its code into ϵ_{victim} in two stages [46]. First, *Conficker* copies pure data into ϵ_{victim}. This data is the path to *Conficker*'s shared library. Then it creates a new thread in ϵ_{victim} executing a function for dynamic loading of shared libraries. The previously injected data poses as input to this function. Hence, *Conficker*'s shared library is loaded into ϵ_{victim}. An observer who is not involved may suppose that ϵ_{victim} has loaded the shared library by itself.

4.3.3 Code Execution Strategy

After having copied code into ϵ_{victim}, ϵ_{inject} triggers the execution of this code. There are several ways to accomplish this. For example, malware may achieve this with the help of debugging APIs (e.g. *CreateRemoteThread* like *Zeus*) or asynchronous procedure calls (e.g. *QueueUserAPC* like *Conficker*) on Windows. Although there are several ways to trigger code execution, we differentiate only two code execution models: *Concurrent Execution* and *Thread Manipulation*. Whereas the original code of ϵ_{victim} continues to execute in *Concurrent Execution*, it does only do so to a certain degree in *Thread Manipulation*.

Basic Definitions

Before we formally define *Concurrent Execution* and *Thread Manipulation*, we present three fundamental definitions. Since these two models are built, for instance, on top of

threads and programs, we present them first. The program of ϵ_{victim} as well as the set of the threads that it theoretically can spawn is presented in Definition 4.9 similar to the one we defined in [15].

Definition 4.9. Let $program_{victim}$ be the program that is executed in the context of ϵ_{victim}. Let $\mathbb{T}_{program} = \{t_{program_1}, ..., t_{program_k}\}$ be the set of all threads that can be spawned by $program_{victim}$ where $k \in \mathbb{N}_{>0}$.

Please note that a pure single threaded program p_1 has only one thread $t_{p_1} \in T_{p_1}$, i.e. $|T_{p_1}| = 1$. Further note that in theory $\mathbb{T}_{program}$ may be infinite. The payload that is injected into ϵ_{victim} as well as the set of its possibly spawned threads is defined in Definition 4.10 similar to the one we defined in [15].

Definition 4.10. Let $payload$ be the code that ϵ_{inject} injects into ϵ_{victim}. Let $\mathbb{T}_{payload} = \{t_{payload_1}, ..., t_{payload_m}\}$ be the set of all threads that $payload$ can possibly spawn where $m \in \mathbb{N}_{>0}$.

Note that $payload$ is handled as data until its execution since code and data share the same memory on the *Von Neumann Architecture* [229]. Finally, Definition 4.11 states the set of currently running threads in the context of ϵ_{victim}.

Definition 4.11. Let $\mathbb{T}_{current} = \{t_i | (t_i \in \mathbb{T}_{program} \vee t_i \in \mathbb{T}_{payload}) \wedge t_i \text{ is running } \}$ be the set of currently running threads of $program_{victim}$ after the injection of $payload$ where $i \in \mathbb{N}_{>0}$.

Definition 4.11 states that after an injection, the threads of a victim process are a union of the original program's threads and the threads that the payload has spawned. We differentiate between two execution strategies depending on the role of the program's threads.

Concurrent Execution

In *Concurrent Execution*, ϵ_{inject} first copies its $payload$ into ϵ_{victim} and then it concurrently executes $payload$ to $program_{victim}$. This is properly defined in Definition 4.12.

Definition 4.12. If $\exists a, b \in \mathbb{T}_{current}$ where $a \in \mathbb{T}_{program}$ and $b \in \mathbb{T}_{payload}$ after ϵ_{inject}'s injection of $payload$ into ϵ_{victim} then this is denoted as Concurrent Execution.

A running process requires at least one thread. After ϵ_{inject} has copied its payload into ϵ_{victim} and has concurrently executed this payload, the number of active threads increases by at least one. Hence, there are now at least two threads that define the

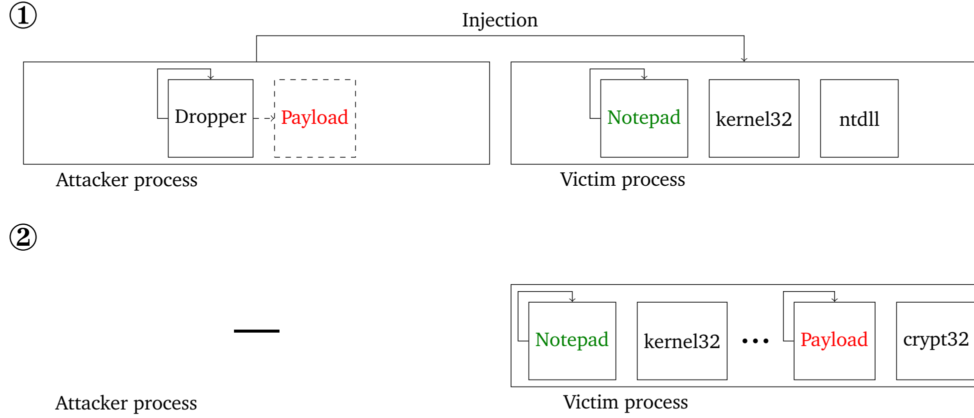


FIGURE 4.2: Concurrent Execution: A dropper injects (ϵ_{inject}) code into a victim process that runs a benign program (ϵ_{victim}). The dropper and the victim comprise at first one thread (self-pointing arrow). After successful injection the dropper terminates and the victim runs a malicious thread concurrently to its original thread. This thread originated in the injected payload.

behavior of ϵ_{victim}, which either belongs to $\mathbb{T}_{program}$ or $\mathbb{T}_{payload}$. This means that the program continues to run while additional malicious code concurrently executes to it. This stands in contrast to the common assumption that only one program is responsible for the behavior of a process.

We outline *Concurrent Execution* in Figure 4.2. It illustrates the state of two processes during an ongoing HBCIA ① and after an HBCIA ②. The left side shows the attacker process ϵ_{inject} and the right side the victim process ϵ_{victim}. In ①, ϵ_{inject} runs a dropper module that holds a pointer to $payload$. ϵ_{victim}'s main program is the editor *Notepad*. Additional modules are mapped into this process space that ensure the editor's functionality. At first, there is only one thread in ϵ_{victim} (depicted by the self-pointing arrow). ① shows the moment when ϵ_{inject} is about to inject its payload into ϵ_{victim}. In ②, ϵ_{inject} is terminated since it is not required anymore. ϵ_{victim} has loaded further modules ($payload$ and the library $crypt32$). An additional thread executes $payload$. This thread concurrently runs to the original thread.

Example: Banking Trojans like *Zeus* utilize *Concurrent Execution* [38]. They inject code into browsers to conduct a *Man-in-the-Browser-Attack*. Thereby, banking Trojans intercept unencrypted banking credentials before they are encrypted by the browser and sent over the network.

Thread Manipulation

Thread Manipulation is the second code execution strategy. It denotes the process of copying $payload$ from ϵ_{inject} into ϵ_{victim} and subsequently executing it via manipulation

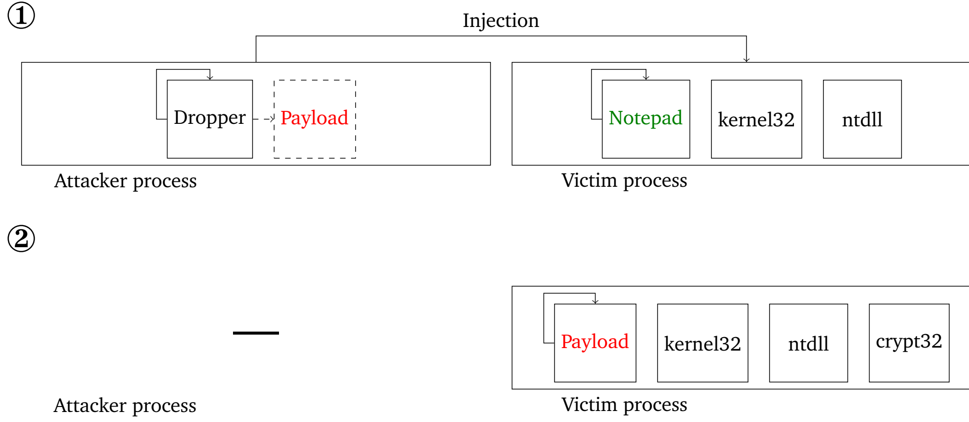


FIGURE 4.3: Thread Manipulation: A dropper (ϵ_{inject}) hollows a program (ϵ_{victim}). The dropper and the victim comprise at first one thread (self-pointing arrow). After successful injection the dropper terminates and the victim continues with one thread. However, now it executes the payload instead of the original program.

of one or several threads of ϵ_{victim}. Hence, one or several threads that executed code of $program$ now execute code of $payload$. Thread Manipulation can render $program_{victim}$ useless. Even though $payload$ runs inside of ϵ_{victim}, the majority of information about the victim does not change, e.g. the process name of the load path of the original binary. At first glance, an HBCIA might seem hard to detect. Contrary to *Concurrent Execution*, $program_{victim}$ does not fully reflect its original behavior. Often it just reflects $payload$'s behavior, which is called *Process Hollowing* in the literature [50], a subclass of *Thread Manipulation*. Definition 4.13 formally defines *Thread Manipulation*.

Definition 4.13. Let $base_p(x)$ be a function that returns the base address of a module x regarding a process space p. Let $start_p(t)$ be a function that returns the start address of an arbitrary thread t in process space p. If and only if the following two assumptions hold

1. $\mathbb{T}_{current} \neq \emptyset$
2. $\exists t \in \mathbb{T}_{payload}$:
   $base_p(program_{victim}) \leq start_p(t) \leq base_p(program_{victim}) + size(program_{victim})$

then this is denoted as Thread Manipulation.

The gist of Definition 4.13 is that there is at least one thread $t \in \mathbb{T}_{payload}$ that originated in $program_{victim}$, i.e. the start address of t lies in the boundaries of $program_{victim}$, which now executes $payload$. This thread was hijacked. This implies that some part of $program_{victim}$ may not work as the author of this program intended.

We depict *Thread Manipulation* in Figure 4.3. It shows the state of two processes during an ongoing HBCIA ① and after an HBCIA ②. Please refer to Figure 4.2 in Section 4.3.3

for a detailed description of the setting. We alter the setting slightly: We assume that ϵ_{inject} started ϵ_{victim}. In ②, ϵ_{inject} terminated since its existence is not required anymore. ϵ_{inject} replaced the original program *Notepad* with $payload$. It loaded an additional module (*crypt32*) and the initial thread of *Notepad* now executes it.

A special case of *Thread Manipulation* is Return-Oriented Programming (ROP) [230]. *ROP* is a data-only exploitation technique [231]. In a nutshell, non-executable memory and code signing prevent the direct injection of code. If an attacker controls the stack and consequently the return addresses, then they can control the control flow of the program. Since they can not directly inject code, they have to chain sequences of code that are already present in the victim program. So far there is no malware family known that implements its whole logic with the help of this technique (see Section 4.4 for an outlook and a discussion on future HBCIA-employing malware families). Though, there are families like *Gapz* that utilize *ROP* [232]. Another special case of *Thread Manipulation* is *Process Hollowing*, where the malware creates the process of its victim in halted mode and directly manipulates the initial thread before any logic of the victim process can execute [50]. This technique allows the malware to hide within the context of a known benign process.

Example: The cyber weapon *Stuxnet* implements *Thread Manipulation* to covertly operate [7]. It starts the trusted Windows process *Local Security Authority Subsystem Service* (lsass.exe) in halted state, manipulates the initial thread, and utilizes this process space for its operations. Hence, *Stuxnet* employs *Process Hollowing*, a form of *Thread Manipulation*.

4.3.4 An HBCIA Algorithm Taxonomy

We identified three fundamental steps of HBCIA algorithms in the previous sections. They are victim process selection, code copying, and code execution. We develop with these fundamental steps a taxonomy to classify HBCIA algorithms.

There are two strategies to select victims: *Targeted Injection* (**TI**) and the *Shotgun Injection* (**SI**). Likewise, there are two strategies for code execution: *Concurrent Execution* (**CE**) and *Thread Manipulation* (**TM**). If we combine the abbreviations of the four strategies (**TI**, **SI**, **CE**, **TM**) then this yields four different HBCIA algorithm classes in total. An example class is the HBCIA algorithm of the computer worm *Conficker*. It comprises *Targeted Injection* (**TI**) and *Concurrent Execution* (**CE**). Therefore, the HBCIA algorithm of *Conficker* belongs to the **TICE** class.

HBCIA Algorithm Class	Example	Operating System
TICE	Hanthie [10]	Linux
TITM	Stuxnet [12]	Windows
SICE	Flashback [11]	Mac OS X
SITM	-	-

TABLE 4.1: Taxonomy of HBCIA algorithms: Three of the four classes can be found in real world malware. Note that there may be **SITM**-employing malware in the future that is fully ROP-based (see Discussion in Section 4.4.)

Table 4.1 lists the four classes of our taxonomy. There are examples for three of the four classes. The *Shotgun Injection* and *Thread Manipulation* algorithm (**SITM**) lacks an example. Up to now, we have not encountered any malware family with a **SITM**-based algorithm. At the time of writing, manipulation of threads of as many processes as possible seems to be a Denial-of-Service (DoS) attack. However, ROP-based malware might exist in the future that utilizes this technique (see Section 4.4) to implement, for instance, code-less banking Trojans.

4.4 The Future of HBCIA-employing Malware

We discuss possible future developments of HBCIA-employing malware in this section. Given the insights of the previous sections, we believe that this aides in anticipating future developments.

We think that the prevalence of HBCIA-employing malware will gradually increase in the future due to the easy access to source code of code-injecting malware projects (e.g. [36–38, 228]). We believe that the most important future development will be *completely* ROP-based malware. ROP (Return-Oriented Programming) is a data-only exploitation technique [231]. Lu et al. [233] showed that malware already leverages ROP in its exploits. In their publication "deRop: Removing Return-Oriented Programming from Malware" they proposed a system for converting ROP sequences into regular shellcode so that it could be handled by malware analysis tools including disassemblers and debuggers. Vogl et al. [234, 235] proposed the idea of data-only malware in 2014. They define data-only malware as "a malware-type that introduces specially crafted data into the system with the intent of manipulating the control flow without changing or introducing new code" [235]. However, they just focused on data-only function hooks. One problem, however, is the long term persistence of malware. They identified four problems of this kind of malware: finding a suitable memory location, protecting against overwrites, resuming the original control flow, and activating the control infrastructure [235]. They addressed these problems in a *Proof-of-Concept* (PoC).

Whereas at the moment the functionality of such malware does not even lie in the range of being comparable to the functionality of today's malware, we suggest that a solution to this restriction could be outsourcing of functionality, for example, to remote servers. This would soften the restriction and allow the accomplishment of complexer operations with little code running on the victim machine. Also, this would complicate a detailed binary analysis due to missing code.

Please note that we can classify future malware families that employ such an approach with our classification taxonomy (see Section 4.3.4). Such malware families would be classified as families that use *Thread Manipulation*, thus possible classes would be **TITM** and **SITM**.

4.5 Conclusion

We have discussed *Host-Based Code Injection Attacks* (HBCIAs) in the context of malware in this chapter. In a nutshell, this attack allows the execution of code within other process spaces. The advantages of HBCIAs explain their popularity: Interception of unencrypted critical information and covert operation, amongst others. However, HBCIAs also come with inconveniences for the malware author such as additional architectural complexity and increased risk of system instability. HBCIA algorithms typically consist of three steps: victim process selection, code copying, and code execution. Based on the definitions of HBCIA algorithms, we have derived a taxonomy for them. Overall, there are four different HBCIA algorithm classes. We assume that the problem size of HBCIAs will not decrease but rather increase in the years to come. ROP-based malware may be just one example of the dormant potential of HBCIA malware.

Quem tudo quer nada tem.

provérbio brasileiro

5

Measuring Host-Based Code Injection Attacks

The last chapter introduced HBCIAs from a theoretical point of view. This should have clarified what an HBCIA is and what its possible consequences are. We have stated that they are popular among todays malware without providing any numbers. We have only provided a wide variety of examples of HBCIA-employing malware families such as *Rovnix*, *Stuxnet*, and *Zeus*. In this chapter, we finally quantify the problem in practice. One concern is to show that we do not deal with isolated cases but rather a common trend in malware. Therefore, we measure the problem size and show that detecting this behavior yields detection of a major share of today's malware. Throughout this section, we address several research questions such as *How prevalent are HBCIAs among current malware families?* and *What are preferred victim processes?*. This section should provide a profound understanding of the problem's practical implications.

We structure the following sections analogously. At first, we formulate the question that guided our research. Then, we present the data set on which and the methodology with which we conducted our investigation. Finally, we discuss the results in detail. At the end of this chapter, we conclude and discuss the results in general by showing the relevance they pose in practice.

Please note that partial results of the presented chapter were published in the seminal paper "Host-Based Code Injection Attacks: A Popular Technique Used by Malware"

Data Set	Year	Publication	Samples	Families
cwsandbox	2007	[52]	6148	unknown
Android Malware Genome Project	2012	[238]	1260	49
Malicia	2013	[239]	11,688	55
Drebin	2014	[240]	5560	179
Microsoft Malware Classification Challenge	2015	[241]	21,741	8
Android Malware Dataset	2017	[242]	24,650	71

TABLE 5.1: Listing of considered publicly available malware research data sets

at the conference "Malicious and Unwanted Software: The Americas" (MALWARE) in 2014 [15] as well as in the seminal paper "Bee Master: Detecting Host-Based Code Injection Attacks" at the conference "Detection of Intrusions and Malware, and Vulnerability Assessment" (DIMVA) in 2014 [16]. The proceedings were published by *IEEE Xplore* and *Springer International Publishing*, respectively.

5.1 Prevalence of HBCIAs in Malware

Malware analysis reports usually follow a common pattern. For instance, they describe the malware packer, a common obfuscation technique that most malware employs to hinder static analysis. Martignoni et al. showcased how packing became a serious problem in the mid-2000s [236]. Whereas in 2003 around one third of the malware samples were packed, in 2008 around 80% of them were packed.

These reports also describe code injections. Take for example reports on Careto (2014) [13], Ponmocup (2015) [237], and Turla (2016) [224]. Even though they usually describe code injections, to the best of our knowledge, there has not been any measurement of the prevalence of HBCIA-employing malware. Based on available data concerning the prevalence of malware families, we could try to indirectly derive an estimation. Four of the top five malware families in 2012 (according to Symantec [209]) employed HBCIAs. They accounted for around one third of all new infection reports in 2012. However, this estimation is not reliable due to the lack of data and may only serve as a first impression of the problem scale.

Due to the fact that a huge share of malware analysis reports include code injections, it is likely to be a widespread problem. Therefore, the first research question that we address is *How prevalent are HBCIAs among current malware?*.

Data Set	R1	R2	R3	R4
cwsandbox	✓	✗	✗	✓
Android Malware Genome Project	✓	✗	✗	✗
Malicia	✓	✗	✗	✓
Drebin	✓	✗	✗	✗
Microsoft Malware Classification Challenge	✓	✗	✗	✓
Android Malware Dataset	✓	✗	✗	✗

TABLE 5.2: Matching of publicly available data sets to our four requirements **R1**-**R4** of Section 5.1. The symbol ✓denotes that a data set fulfills a certain requirement, the symbol ✗denotes that a data set does not fulfill a certain requirement.

5.1.1 Data Set

To the best of our knowledge, there had not been any longitudinal study of the prevalence of HBCIA malware. Therefore, there were no requirements for a data set to conduct such a study. Consequently, we first defined our requirements for such a data set.

The quality of an investigation greatly depends on the underlying data set. We required a representative share of malware spread over a long period of time to measure the prevalence of HBCIA-employing malware. Compiling a malware data set is not trivial. There is neither a common creation process nor an agreement on its content. At the time of writing, compiling such a data set is an open problem, which also overlaps with the malware labeling problem [243].

Requirements

Rossow et al. [244] discussed best practices to set up (dynamic) malware experiments. However, they primarily focused on how experiments should be conducted rather than the data itself. Nevertheless, they gave advice on how to build a data set. Their most relevant advice was:

A1 There should be no goodware in the data set.

A2 The malware families should be balanced in the data set.

A3 The data set should contain only relevant malware families.

We adhered to **A1**. This was our requirement **R1**. However, we discarded **A2** and **A3**. It is natural that some families are more prevalent than others, e.g. since they are spread more aggressively. Therefore, they should also be represented with their actual weight in the data set (**A2**). Hence, we refrained from balancing the families to

preserve these weights. Furthermore, **A3** is subjective: How do we measure relevance of a malware sample or a family in general? We could utilize its population estimation. But then targeted malware would not be relevant anymore. It usually comprises very small populations. Finding the right metric to prove relevance is not trivial. Consequently, we refrained from explicitly measuring the relevance.

We required the data to be captured within a closed period of time in order to conduct a longitudinal study. This period should be long enough to be able to reliably estimate the HBCIA prevalence. It is certain that the longer the period, the better. But due to problems such as data set size and limited computational resources, we are restricted as individual researchers to a small part of all known malware. We are facing less than 25 years of malware development on Windows NT. If we insist that the period should be longer than one year then this would still be more than $\frac{1}{25}$ of this history. To the best of our knowledge, the computer viruses of the 1990s did not utilize HBCIAs. So the history of HBCIAs is even younger. The samples should also be evenly distributed over this period so as to there is no bias due to a skewed distribution. For instance, a huge cluster of samples on just a couple of dates and a tiny fraction spread over the rest of the period would not allow us to make a reliable statement about the behavior within this period. As a result, **R2** required that the malware samples should have been captured within a 1+ year long, closed period with samples distributed evenly over the period on a per day basis.

To make a statement about malware in general, we required a significant number of samples. There are millions of new samples each year (e.g. [4]). It would not be sufficient to make a statement about a couple of hundred samples. Even though significant number of samples is a vague statement, we wanted to surpass the number of samples s of previous research data sets, i.e. $s \geq 25,000$. Therefore, requirement **R3** required a significant number of malware samples to be in the data set. Please note that we are not making a statement about malware families at this point. The number of malware families is by orders of magnitude smaller than the number of samples [245]. But for this measurement, the concept of malware family was not relevant.

Microsoft Windows is the main target of malware when measured in absolute numbers (see for instance [4]). It is also the market leader in desktop operating systems (as of November 2017) [8]. Furthermore, there are only few HBCIA-employing malware families for other operating systems known. Therefore, **R4** required only Windows malware. We are considering further measurements for other platforms as future work (see Section 8.2).

To summarize our requirements:

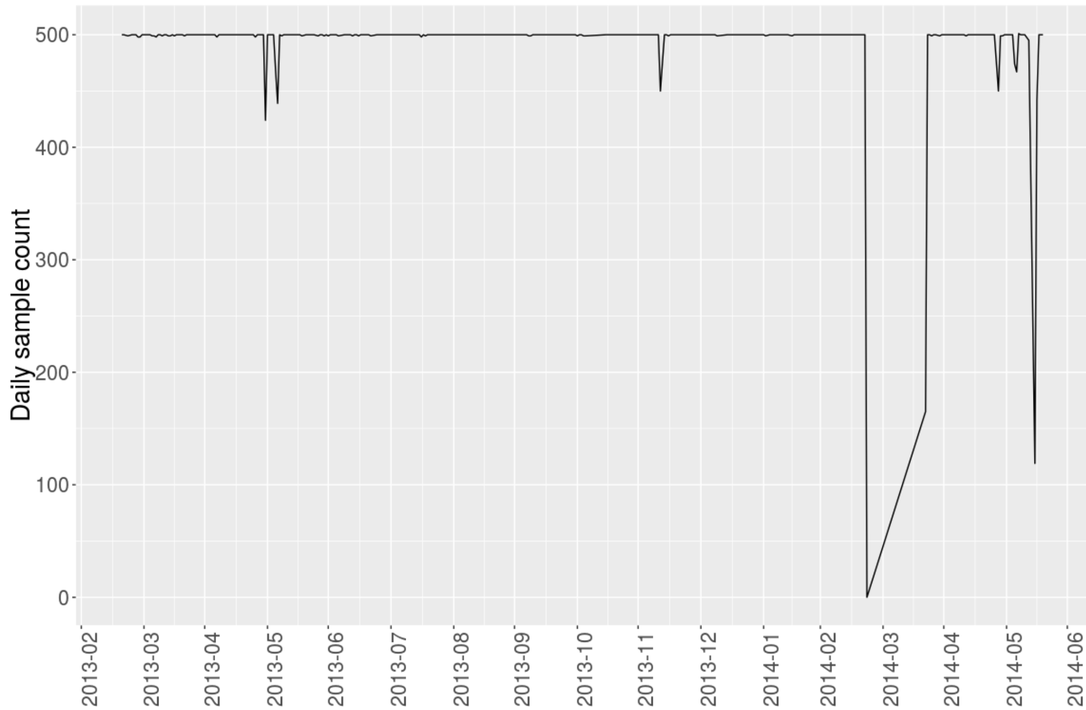


FIGURE 5.1: Daily number of samples from 2013-03-21 until 2014-06-19 of the *VirusSign* data set. *VirusSign* provided 500 samples daily. However, there were periods where no samples were available, e.g. in March 2014.

R1 There should be no goodware in the data set.

R2 The malware should be captured within a 1+ year long, closed period with samples distributed evenly over the period on a per day basis.

R3 There should be a significant number of samples s, ($s \geq 25,000$).

R4 There should only be Windows malware.

There were several publicly available research data sets, which we matched to our four requirements. Table 5.1 summarizes them. They were published (in the most cases) as results of a seminal papers at ranked computer security conferences. The number of samples greatly varied in them. The *Android Malware Genome Project* contained the least (1260) and the *Android Malware Dataset* contained the most samples (24,553). All data sets targeted either Windows or Android.

We have matched the data sets to our four requirements in Table 5.2. None of them matched all requirements. To the best of our knowledge, there was no publicly available data set that matched our requirements. Therefore, we compiled our own.

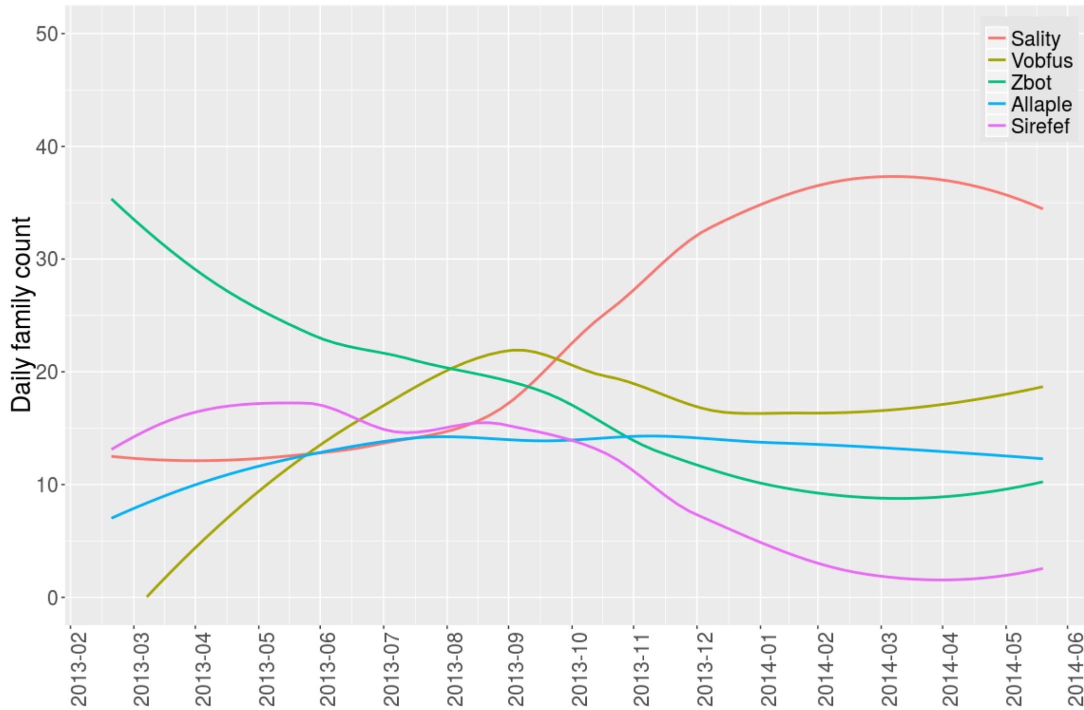


FIGURE 5.2: Top five families over time from 2013-03-21 until 2014-06-19 of the *VirusSign* data set. This data set reflects, for instance, the takedown of the *Sirefef* botnet in December 2013.

Data Set Compilation and Discussion

VirusSign offers a huge collection of malicious samples, amongst others, for research [246]. They confirm the maliciousness of their samples with automatically generated reports. This company regularly offers (mostly daily) a free data set for malware research that consists of around 500 malware samples. They randomly choose these samples from all their daily samples. We aggregated their data sets over 15 months to create a data set that models the malicious threat landscape. The resulting data set consisted of 162,850 malware samples. These samples were collected between 2013-03-21 and 2014-06-19. Due to this rather long period, we assume that it adequately represents the threat landscape between 2013 and 2014. Figure 5.1 shows the sample distribution by publication date. Unfortunately, the data set was not continuous. There are periods where no samples were available. These are [2013-09-05, 2013-09-21], [2013-11-06, 2013-11-14], [2014-03-06, 2014-03-12], and [2014-03-25, 2014-04-21]. Please note that our data set considered more malware samples than other popular data sets such as the *Malicia* set [239] or the *cwsandbox* set [52].

We queried the antivirus scanning service *VirusTotal* [247] for each sample of the data set. None of *VirusTotal*'s 57 scanner (as of October 2016) had a perfect detection rate. Many of them detected more than 99% of the samples as malware. Please note that

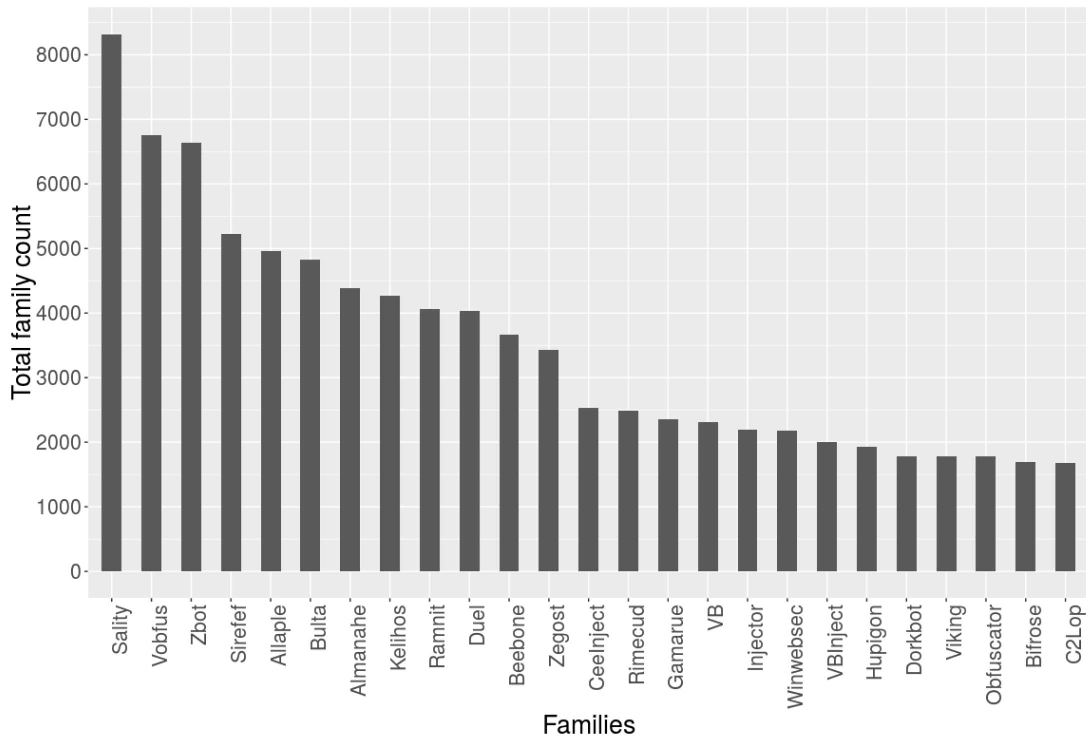


FIGURE 5.3: Top 25 families from 2013-03-21 until 2014-06-19 of our data set.

we queried *VirusTotal* in August 2016, this was two years after the malware emerged. Since this malware data set had been distributed in the antivirus industry, the antivirus companies had sufficient time to add signatures to their databases. The scanner with the highest detection rate outputted many generic labels so that it was impossible to evaluate its results. The scanner *Microsoft Security Essentials* (MSE) outputted few generic labels and it appears that this scanner tries to present the user an accurate classification. Therefore, we chose the results of MSE as our ground truth. Please note that this is not an exact ground truth due to the inaccurate detection of antivirus programs [248]. MSE has a detection rate of 96.47%. This means that it detected 157,078 of the 162,815 samples as of October 2016.

Malware labeling schemes are complex. The labeling of MSE reflects the target OS, the architecture, the family, the variant, and the malware class, e.g. *TrojanSpy:Win32/Ursnif.HN*. We stripped everything, except the family name, from the labels. Given the example above, this yields *Ursnif*. *MSE* detected 1733 distinct families. Figure 5.3 shows the top 25 malware families of the data set. The top family was the virus *Sality*, which had a share of approximately 4% of the data set. It was followed by *Vobfus* (also known as *Ponmocup*) and *Zbot* (also known as *Zeus*), which are credential stealers. In general, the distribution of families seems to be in line with reports from the antivirus industry (e.g. [237, 249]).

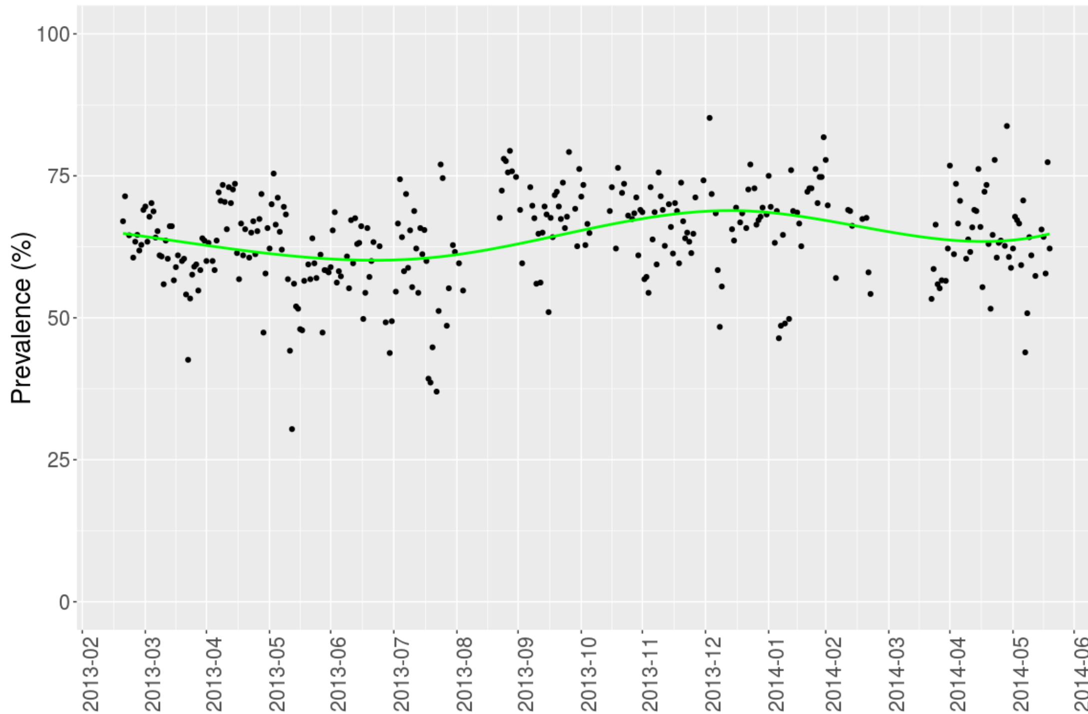


FIGURE 5.4: Weekly percentage of HBCIA-employing malware from 2013-03-21 until 2014-06-19 as a scatter plot.

Figure 5.2 plots the daily number of samples of the top five families over time. Interestingly, the data reflects the takedown of the *Sirefef* botnet (also known as *ZeroAccess*). Microsoft, Europol, and the FBI took down the botnet in December 2013 [250]. Figure 5.2 shows a drop of *Sirefef* samples at the end of November 2013/beginning of December 2013. We could not find any explanation why the share of the family *Sality* significantly increased after October 2013. We suspect that *Sality* benefited from the *Sirefef* takedown. Both *Sirefef* and *Sality* are peer-to-peer-based families. At the beginning of the period, the family *Zbot* had the highest share of the five families. However, it significantly decreased during the period. Maybe cyber criminals turned their backs on *Zbot* and employed other banking Trojans instead, e.g. its successors such as *Kins* and *Citadel*. Furthermore, we observe a continuous increase in the family *Vobfus*. This gain is also in line with reports that state that this family was one of the most prevalent families of its time [237].

We conclude that our data set matches all stated requirements and that it seems to be a reasonable representation of the malware threat landscape of the years 2013 and 2014. It depicts notable events like the *Sirefef* takedown in December 2013 and it comprises more samples than any other publicly available scientific data set. Therefore, it is adequate to measure the distribution of HBCIA-employing malware in the wild.

5.1.2 Methodology

A decision should be made for each malware sample of our data set whether or not it employs an HBCIA in order to measure the prevalence of HBCIAs. However, it is undecidable whether a given program implements HBCIAs or not. This comes from *Rice*'s theorem. In a nutshell, *Rice*'s theorem is a generalization of Turing's *Halting Problem*. It states that it is undecidable whether an algorithm computes a partial function that has a non-trivial property [251]. In our case, the non-trivial property was an HBCIA. Hence, we could only try to approximate this problem.

One way to do this is statically analyzing the code of the sample but the *Unpacking Problem* complicates this [252]. Another way is executing the malware and logging its behavior. Then, we can analyze these logs and detect the injection behavior, if it occurs. The advantage of this approach is that we circumvent the *Unpacking Problem*. Environment sensitive malware, i.e. malware that refuses to run in analysis environments [253], may render this approach infeasible. We can reduce the amount of malware that refuses to run in the analysis environment, if we sufficiently harden the analysis environment [254]. Given the fact that most malware comes in a packed form [55], we opted to utilize dynamic analysis and analyze the samples in a sandbox. Then, we processed the sandbox reports to detect code injection behavior.

VirusSign operates a sandboxing system called *VSAMAS* [246]. They distribute a *VSAMAS* log with each malware sample in their data sets. *VSAMAS* uses Windows XP SP3 32 bit internally. Even though Windows XP is not a recent version, it is based on the Windows NT-platform that is still the core of Windows. Newer versions of Windows are based on this platform, i.e. programs written for Windows XP are also compatible to Windows 10. Furthermore, Windows XP still accounted for more than 6% of all active operating systems in August 2017 [8]. It exhibited almost the same market share as Windows 8.1. This still surpassed the accumulated market shares of macOS. Because of this, malware continues to target this platform.

We parsed each sandbox report and searched for sequences of suspicious API calls that suggested the occurrence of an HBCIA. An example is the API call sequence *AllocateVirtualMemory*, *WriteMemory*, and *CreateRemoteThread*. Please note that we surveyed the occurrence of HBCIAs (e.g. **TICE**) regardless of the employed injection technique. Our measurement is a *lower* bound of HBCIA-employing malware. As stated before, we may have missed some code injections due to environment sensitive malware.

5.1.3 Results

Nearly two thirds (63.94%, 104,139/162,850) employed HBCIAs. Figure 5.4 plots the weekly prevalence of code-injecting malware. We interpolated the underlying curve using a B-spline of degree 5 (green curve). The majority of the weeks closely spreads around the absolute mean of 63.94%. The data is bounded below by 48.43% and bounded above by 78.5%. There is neither an ascending nor a descending tendency in the slope. Therefore, we assume that code injections were a stable feature of the samples in this period.

We consider the result as a lower bound. Evasive techniques like sandbox detection are wide-spread [47, 253]. Therefore, we assume that some HBCIA-employing samples were not successfully processed. As a consequence, we conclude that there were even more samples that employ HBCIAs attacks. This result is also in line with the initial indication given by the Symantec report [209] that showed that four of the five top families regarding infection count employed HBCIAs in 2012. Our measurement suggests that HBCIAs are a relevant problem to security researchers due to their impact in sheer numbers. This means that the detection of HBCIAs implies the detection of a great share of today's threat landscape.

5.2 Preferred Victim Processes

The previous section has shown the significant share of HBCIA-employing malware. We assume that the victim processes are not evenly distributed since not all HBCIA-employing malware families employ *Shotgun Injections*. Consider the Windows process *explorer.exe* as an example. There is a clear tendency that many malware families inject their code into this process (e.g. [36, 38, 228, 255]). One reason for this behavior is the availability and the accessibility of this process [255]. Therefore, it would be interesting to determine the distribution of the victim processes.

The insights provided by the distribution of the preferred victim processes is twofold: First, malware analysts can prioritize the processes they investigate to speed up the analysis process in general. Second, security tools can mock processes that are more likely to be attacked to increase the probability of attacks (see Chapter 6).

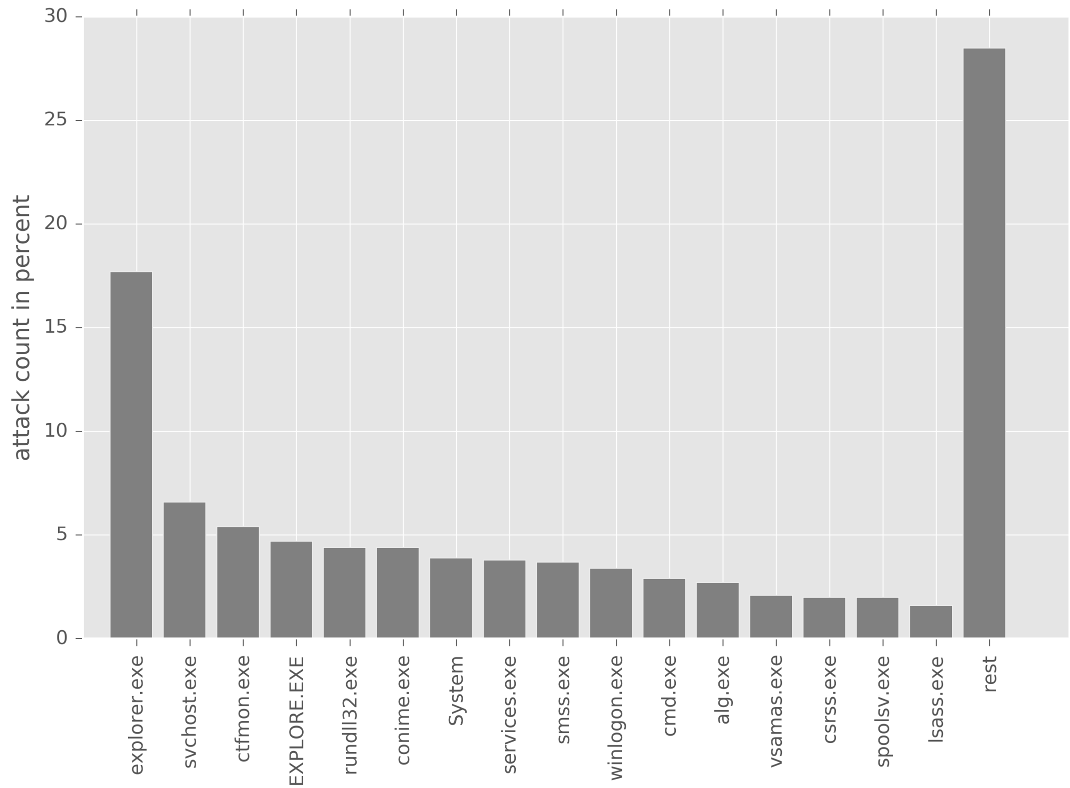


FIGURE 5.5: Relative distribution of victim processes on Windows XP. Most HBCIAs targeted *explorer.exe*, which is a Windows system process that always runs.

5.2.1 Data Set & Methodology

The data set of this section was based on a subset of Section 5.1. In this section, we have determined that nearly two thirds of the data set employed HBCIAs. In these 104,139 cases, *VSAMAS* logged the attacker process and the victim process with each detected HBCIA. Hence, we could derive the distribution of the victim processes from this data. This yielded a good overview of the victim processes in practice, given the fact that this data set adequately represented the malware threat landscape of the years 2013 and 2014 as discussed in the previous section.

5.2.2 Results

Figure 5.5 depicts the distribution of the victim processes. The main target was *explorer.exe* as expected. Malware targeted it in 17.7% of all HBCIAs. Furthermore, 16 processes shared almost three quarters of all attacks. These processes were all system processes (except *vsamas.exe*), which are found on every Windows XP installation.

This shows that the attacks were not evenly distributed as assumed before the evaluation. System processes were preferred over other processes that are not default Windows programs. There are several explanations. One explanation for this behavior is that the malware author can be sure that these processes are available (e.g. *explorer.exe* [255]). Another explanation is that some of these processes are whitelisted by personal firewalls (e.g browsers like *iexplorer.exe*) and malware utilizes them as communication gateway (see Section 4.2.1). A third explanation is that many processes are more likely to be targeted due to the data they handle (e.g. browsers in the case of banking Trojans). We recommend detection systems to focus at first on the top five processes: *explorer.exe*, *svchost.exe*, *ctfmon.exe*, *iexplorer.exe* and *rundll32.exe*. This should increase their chances to encounter HBCIAs.

5.3 Family Feature HBCIAs

HBCIAs offer many advantages to malware authors (see Section 4.2). Yet, their utilization is a design decision that is taken early on during the development, e.g. to conduct *Man-in-the-Browser* attacks. Therefore, they form an integral part of the malware. This decision significantly influences the malware's code base, for instance, due to synchronization of several infected processes.

Hence, we assumed that malware authors are unlikely to refrain from code injections. They are unlikely to change the method or even completely remove it, since this design decision is taken early on once the objectives of the malware are known. Furthermore, this should also hold true in the case of derived malware that is based on a code leak, e.g. the banking Trojan *Kins* [228], which derives its code base from *Zeus* [38].

Therefore, we assumed that HBCIAs are an inherent malware family feature. This means that versions and variants of the same family but also derivatives that have large parts of the code base in common neither remove this feature nor change the code injection method over time. If this assumption were true then as a consequence it would be sufficient to study one representative of a malware family to study the family's HBCIA behavior. In the following, we corroborated our assumption with a longitudinal study of eight code injecting malware families.

5.3.1 Data Set

Unfortunately, the data set of Sections 5.1 and 5.2 was not adequate to conduct a longitudinal study of the code injection behavior of several families. First, there was no exact

Data Set	R1	R2
cwsandbox	✗	✗
Android Malware Genome Project	✗	✗
Malicia	✗	✗
Drebin	✗	✗
Microsoft Malware Classification Challenge	✓	✗
Android Malware Dataset	✗	✗

TABLE 5.3: Matching of publicly available data sets to our two requirements of Section 5.3 to a corpus for corroborating the family feature hypothesis.

classification of the data set. We needed to be absolutely sure that we inspected variants and versions of a certain family. Note that labels given by an antivirus software are not exact [248]. Second, the history of a malware family may date back more than ten years. For instance, the *Zeus* family is at least ten years old (as of November 2017) [33]. We could not model this with this data set since it comprised only 15 months of data.

To the best of our knowledge, there had been no prior longitudinal study of the injection behavior of malware families. Therefore, there were no requirements to a data set to study this behavior. Consequently, we defined two requirements in order to compile our own data set in the following.

Requirements

The objective of this section is to measure the code injection behavior of several families to show that neither different variants nor different versions of a malware family change this behavior over time. Therefore, our first requirement **R1** wass that the data set comprised only malware families that employ HBCIAs.

To make a statement about the HBCIA behavior of a malware family, we needed to follow its evolution. As every other software, malware is incrementally developed. There may be significant differences between two versions that lay several months apart. Furthermore, malware authors may change significant parts of a malware when its tasks change. We required several versions spread over a longer period of time to reliably make a statement about the family's (HBCIA) evolution. Therefore, **R2** stated that the families of the data set were required be active at least one year and that they comprised several versions so as to we could follow their development and possible behavioral changes. Given the history of modern Windows malware, which started with the release of Windows XP, one year poses a substantial share of this history.

To summarize our requirements:

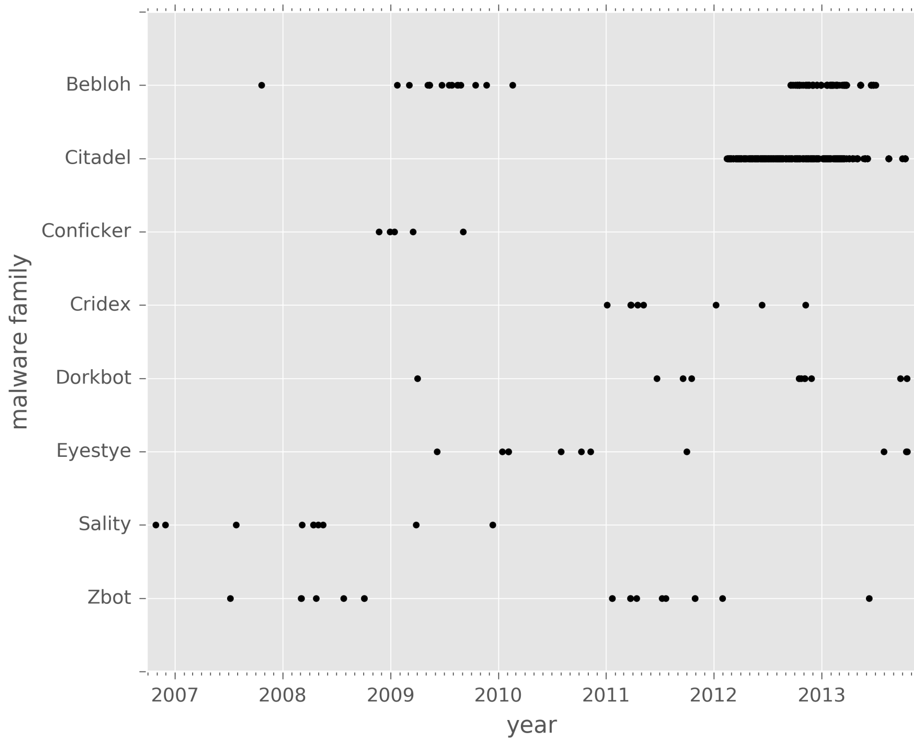


FIGURE 5.6: The temporal distribution of the eight HBCIA-employing malware families of our data set ranging from 2007 to 2013.

R1 The data set contains only malware families that exhibit HBCIA behavior.

R2 The malware families should have been active at least for one year and comprise several versions.

We matched the publicly available malware data sets that we have presented in Section 5.1 to our two requirements. Table 5.3 presents the results. There was no publicly available data set that matched our requirements. Especially, since they did not contain meta information like the sample date. Therefore, we had to create our own.

Data Set Compilation and Discussion

Our data set consisted of eight distinct code injecting malware families (**R1**). Even though we face a flood of malware samples today, the number of malware families is actually by orders of magnitude smaller [245]. We gathered versions as well as variants of each version throughout several years (2007-2013) (**R2**). The exact numbers are given in Table 5.4.

Tables 5.4 summarizes the data set. It contained 32,514 samples of eight malware families in total. *Citadel* comprised the lion's share of the data set. However, this did not affect the results since we examined each family separately. We list the number of samples, the number of versions, as well as the period that lies between the first and the last version for each family. We determined this period with the help of VirusTotal (first time seen) [247], except for *Urlzone* and *Citadel*. In these cases, we had in-house unpackers extract the timestamp of the original PE (Portable Executable).

Even though packers may forge many values of their executables, this is typically not the case with the original executable that they pack. The malware authors may forge the meta data of the original executable, e.g. which is often the case with espionage malware [256]. However, we have internal data that suggests that this was not the case with both aforementioned families. Figure 5.6 plots the temporal distribution of the samples. *Citadel* was the latest family. It comprised the most samples, which were therefore closely scattered in 2012 and 2013. The other families like *Zeus* operated longer, e.g. for more than five years.

5.3.2 Methodology

At first, we conducted a manual analysis of several family members to understand the HBCIA algorithm. Based on this knowledge, we extracted a characteristic API call sequence that describes this algorithm. Next, we implemented a behavior analysis processing module for the sandbox system *Cuckoo* [257] and automated the execution of all samples. This module processed the captured API call sequences and matched them to the known HBCIA algorithm sequences, e.g. *OpenProcess*, *WriteProcessMemory*, and *CreateRemoteThread*. The arguments for choosing dynamic analysis were the same as in Section 5.1.

We utilized the sandbox *Cuckoo* [257] to record the API call sequences. This sandbox is the de facto standard because it is enhanceable and accessible owing to its open source license. At its heart, *Cuckoo* employs *VirtualBox* [72] with Windows XP SP3 32 bit. The same arguments regarding the choice of this Windows version hold true as in the preceding Section 5.1. The execution environment had no Internet connection. This should not have influenced the results due to the assumption that HBCIAs take place before the malware contacts its command and control server. Numerous source code leaks of malware corroborate this behavior (e.g. [36–38, 228]). We hardened the execution environment to cope with environment sensitive malware [253].

Malware Family	Samples	Versions	Date of First/Last Sample
Citadel	31713	18	2012-02-14/2013-10-10
Conficker	5	5	2008-11-22/2009-10-31
Cridex	12	4	2011-01-04/2012-11-07
Dorkbot	21	7	2009-04-01/2013-10-16
Sality	20	6	2006-10-27/2013-07-02
Spyeye	12	4	2009-06-06/2013-10-17
Urlzone	701	63	2007-10-21/2013-07-02
Zbot	30	10	2007-07-07/2013-06-09
Total	**32514**	**117**	**2006-10-27/2013-10-17**

TABLE 5.4: Summary of the data set for the family feature investigation of Section 5.3.

5.3.3 Results

The results corroborated our assumption. All families neither removed nor changed their injection behavior. As a consequence, we consider HBCIAs as an integral part of malware families that employ them. Especially, it is invariant over different versions and variants of such families.

We corroborated that the HBCIA algorithms of *Zeus* [38] and its offspring *Citadel* are identical via a binary comparison. This backs our hypothesis that malware authors prefer to quickly build a new malware on a leaked source base. They refrain from fundamental changes due to the lack of time or the lack of understanding.

5.4 Prevalence of HBCIA Algorithms

In Sections 5.1 and 5.2, we measured the prevalence of HBCIAs and scrutinized the preferred victim processes. This showed that victims are not evenly distributed. Hence, there have to be different selection strategies in practice.

We derived a taxonomy for HBCIA algorithms in Section 4.3.4 that comprises four classes: **TICE**, **TITM**, **SICE** and **SITM**. However, so far there has been no measurement of the distribution of malware families to these classes. The objective of this section is to address the question *What is the distribution of malware families to the different HBCIA algorithm classes?*

5.4.1 Data Set

First of it all, the data sets of Sections 5.1 and 5.2 as well as Section 5.3 were not suitable to be utilized in a qualitative evaluation that should comprise many different, clearly identified malware families in order to study their HBCIA behavior. The first data set of Sections 5.1 and 5.2 did not comprise an exact classification of the malware samples. Even though we approximated this with malware labels provided by an antivirus software, this was not exact. The second data set of Section 5.3 only contained eight families with many versions and variants. It was not suitable because we required more family representatives to estimate the distribution to the HBCIA algorithm classes. There had not been any previous work focusing on this question that defined any requirements for a data set. Consequently, we first had to define our requirements for a data set.

Requirements

We had two requirements for a data set for measuring the distribution of malware families to HBCIA classes. There are many malware classes including but not limited to *computer worms*, *banking Trojans*, and *viruses* that can be encountered in the wild. In order to estimate the prevalence of the four different HBCIA algorithm classes, we needed to scrutinize as many different HBCIA-employing malware families as possible. Please note that an HBCIA is a malware family feature and hence it is sufficient to study one representative of a malware family in order to study the family's HBCIA behavior (see Section 5.3). Furthermore, it is important that this behavior is confirmed for every member of the data set because we explicitly studied this behavior. Therefore, our first requirement **R1** required the data set to contain many different, clearly identified malware families of all kind of types that were confirmed to use HBCIAs. Our second requirement **R2** required the malware families to target Windows, since it is still the main malware target, when measured in absolute numbers (see for example [4]).

Table 5.5 summarizes the matching of the two requirements to the set of publicly available research data sets (see Table 5.1). None of them matched our requirements. Hence, we decided to create our own to answer our research question.

Data Set Compilation and Discussion

We collected representatives of HBCIA-employing malware families. We only needed to examine one representative per malware family to determine the family's HBCIA algorithm, since HBCIAs are a family feature (see Section 5.3). The data set comprised

Data Set	R1	R2
cwsandbox	✗	✓
Android Malware Genome Project	✗	✗
Malicia	✗	✓
Drebin	✗	✗
Microsoft Malware Classification Challenge	✗	✓
Android Malware Dataset	✗	✗

TABLE 5.5: Matching of publicly available data sets to our two requirements **R1** and **R2** of Section 5.4.

40 malware families, for instance, banking Trojans (*Urlzone*), cyber espionage malware (*Stuxnet*), malware droppers (*Matsnu*), and click fraud malware (*Sirefef*). The considered malware families are listed in Appendix B.

5.4.2 Methodology

To determine the HBCIA class of a sample, we were required to scrutinize its behavior. The sequence of API calls that a sample exhibits during its execution (partially) describes its behavior. Given this sequence, we could derive the HBCIA class of a malware. For instance, the banking Trojan *Tinba* [36] utilizes the API sequence *CreateToolhelp32Snapshot*, *Process32First*, *Process32Next*, *lstrcmpiA*, *OpenProcess*, *WriteProcessMemory*, and *CreateRemoteThread*. First, it compares all running processes (*CreateToolhelp32Snapshot*, *Process32First*, *Process32Next*) to a process name it wishes to attack (*lstrcmpiA*). Then, it copies its code into the victim process (*OpenProcess*, *WriteProcessMemory*) and concurrently executes it (*CreateRemoteThread*). Hence, it falls into the class **TICE** (Targeted Injection/Concurrent Execution).

We analyzed each representative with Cuckoo sandbox [257] to determine its HBCIA API call sequence (see last section). The sandbox monitored API calls and outputted the sequence. From this data, we derived the class of the sample as shown in the example above. In the case of environment sensitive malware, we manually analyzed samples to extract the API call sequence. The arguments for choosing dynamic analysis were the same as in Section 5.1.

5.4.3 Results

The majority of the samples employed a **TICE** algorithm (Targeted Injection/Concurrent Execution). The rest fell into the two classes **TITM** (Targeted Injection/Thread Manipulation) and **SICE** (Shotgun Injection/Concurrent Execution). No family employed a

HBCIA Algorithm Class	Malware Families	Percentage
TICE (Targeted Injection/Concurrent Execution)	23	57.5%
TITM (Targeted Injection/Thread Manipulation)	8	20.0%
SICE (Shotgun Injection/Concurrent Execution)	9	22.5%
SITM (Shotgun Injection/Thread Manipulation)	0	0.0%

TABLE 5.6: The distribution of malware families to HBCIA algorithm classes in absolute figures and percentage. All classes are populated except the **SITM** class.

SITM algorithm (Shotgun Injection/Thread Manipulation). Table 5.6 summarizes our findings.

The result shows that *Targeted Injections* (**TICE** + **TITM** = 77.5%) were more popular than *Shotgun Injections*. There are two reasons for this. First, this allows malware to skip crucial system processes. Crashes in system processes may result in total system failure, which could lead to detection of the malware by the user. Second, selection of target processes yield less usage of system resources and less irrelevant data. For instance, many banking Trojans just inject themselves into browsers, and Point-of-Sale (PoS) malware just injects its code into processes of PoS programs. Furthermore, the result shows that *Concurrent Execution* (**TICE** + **SICE** = 80%) was more popular than *Thread Manipulation*. A reason for this might be that the injected code requires the original code to be run to manipulate it, e.g. in the case of credential stealers.

5.5 Conclusion

Today's malware widely utilizes HBCIAs. Almost two thirds of our data set showed this behavior. An HBCIA is therefore a reliable *Indicator of Compromise* (IoC) for malware. Malware analysts and security tools should prioritize the processes that they examine based on our results. For instance, *Explorer.exe* exhibits a high attack probability. In addition, it is a behavior that is unlikely to change over variants and versions of a malware family since it is an inherent family feature. We only encountered malware implementing **TICE**, **TITM**, or **SICE** algorithms. The fourth class **SITM** was absent. In general, malware authors seem to favor *Targeted Injections* to *Shotgun Injections* and *Concurrent Execution* to *Thread Manipulation*. The victims were not evenly distributed, since not all HBCIA-employing malware families implemented *Shotgun Injections*.

Based on the results of this chapter, it can be seen that HBCIAs are a fundamental part of a much bigger problem called malware. As a consequence, HBCIA detection could lead to detection of a significant share of current malware and therefore should be pursued additionally to traditional antivirus techniques.

Developers, developers, developers, developers.

Steve Ballmer

6 Bee Master: Detecting Host-Based Code Injection Attacks at Runtime

The previous two chapters have investigated the phenomenon of *Host-Based Code Injection Attacks* (HBCIAs) in the context of malware. This has provided an overview of the different theoretical models and the scale of the problem. Current solutions are not well suited for malware and also do not focus on the OS-independent detection of HBCIAs (see Chapter 3). Therefore, we developed two solutions for the (dynamic and static) detection of HBCIAs. Whereas the dynamic solution focuses on end user protection, the static one aids malware and forensic analysts.

In this chapter, we present the first of our two approaches called *Bee Master*, a system to dynamically detect HBCIAs. In a nutshell, we apply the honeypot paradigm to OS processes to detect HBCIAs. *Bee Master* creates regular operating system processes that are vulnerable to these attacks. These processes mimic programs like browsers or chat clients. It monitors them for changes associated with HBCIAs. These changes include the number of threads and memory pages. Since the behavior of these processes is apriori known, any deviation is suspicious. OSes utilize processes to manage concurrency. Hence, the main idea applies to many operating systems including Windows and Linux. As a consequence, our system neither requires modifications of the OS nor hardware as previous systems require (e.g. [23, 162, 163]).

Please note that partial results of the presented chapter were published in the seminal paper "Bee Master: Detecting Host-Based Code Injection Attacks" at the conference "Detection of Intrusions and Malware, and Vulnerability Assessment" (DIMVA) in 2014 [16]. The proceedings were published by *Springer International Publishing*.

6.1 Methodology

This section presents our approach *Bee Master* to dynamically detect HBCIAs. It applies the honeypot paradigm to OS processes to detect code injections. Amongst other advantages, this allows the detection of HBCIAs independently of the operating system.

Before we present *Bee Master* in detail, we elaborate the requirements for our approach. Then, we provide a rough overview of it and its two components: the *Queen Bee* and the *Worker Bees*. Subsequently, we detail these two components in the next two sections. Next, we discuss the proof of concept implementation and the limitations of *Bee Master*. Finally, we describe possible ways of evasion and countermeasures.

6.1.1 Requirements

We have shown in Chapter 3 that there are systems to dynamically detect HBCIAs. However, they have weak points, e.g. dependence on low-level APIs or the need of special hardware. Their authors did not state any requirements when they presented their proposals.

An HBCIA is a dynamic behavior that a malware exhibits during runtime. Even though we might be able to statically detect this behavior given a malicious binary, this would be limited. First, we would face the unpacking problem and therefore static analysis is not feasible in our case. For a detailed discussion of this problem see Section 5.1. Second, we can not decide whether a program conducts an HBCIA or not. As we have stated in Section 5.1, this follows *Rice*'s theorem. In our case this is especially owing to the fact that we may face dynamic code generation, which also is related to the first limitation. Even though we can not decide this beforehand, we can still observe the program and detect this behavior if it executes it. Consequently, we require our system to detect such code injections at runtime (**R1**).

The diversification of operating systems is a severe problem for malware detection. There is no single OS that all malware targets. This also holds true in the case of HBCIA-employing malware. In Chapter 4, we have shown that operating systems like Windows, Linux, macOS, and Android are all vulnerable to HBCIAs. A solution that

depends on a specific OS would not be portable. This would imply that each OS would need its own solution, which would be costly in means of time and money. Hence, an HBCIA detection system should not rely on OS-specific details. As a consequence, **R2** states that the approach should be OS agnostic to be easily portable.

Malware utilizes low-level APIs like *NtOpenProcess* and *NtCreateRemoteThread* to conduct HBCIAs. Such low-level APIs are often subject to change and may rapidly break systems that rely on them. Take as examples antivirus real-time protection [258] and also malware itself. Yet, many detection systems just instrument a limited set of low-level APIs and/or special system calls (e.g. Sun et al. [20] and Hanel [169]) to detect HBCIAs. However, there is more than one way to conduct HBCIAs and relying on a limited set of low-level APIs is not a proper solution to catch them all. As a result, **R3** requires that the system abstracts from low-level APIs as well as special system calls and that it relies on high-level concepts instead, which properly describe HBCIAs.

Operating systems are complex buildings. A small architectural change may influence great parts of the code base. This is the reason why many ideas require years to be incorporated in an OS, because it takes significant time for OS vendors to develop, test, and deploy new systems that introduce fundamental changes to the OS itself. Take for example *Control Flow Integrity* (CFI). CFI was only recently introduced to Windows 10 as *Control Flow Guard* [259]. It took more than ten years from the idea to the deployment. However, the fight against malware is fought right now. Consequently, **R4** states that the approach should not modify the operating system's internals.

We assume that we are dealing with malware that comes only in binary form (see also Chapter 2). This is a reasonable assumption since the absence of source code increases the complexity of the analysis. Even though there are several malware code bases publicly available due to source code leaks, the vast majority of malware is only known in its binary form. As a consequence, **R5** assumes that the source code of the malware is not available.

In the following, we summarize our five requirements **R1** to **R5**:

R1 Detection of *Host-Based Code Injection Attacks* at runtime

R2 Operating system agnostic

R3 High abstraction level, independent of low level OS details

R4 No modification of the operating system is required

R5 Works in the absence of the malware's source code

Approach	R1	R2	R3	R4	R5
Barrantes et al. [116]	◐	●	○	○	●
Buescher et al. [22]	◐	○	○	●	●
DefenderATP [205]	●	○	○	○	●
Forrest et al. [173]	●	●	○	●	●
Hanel [169]	●	○	○	●	●
Korczynski et al. [172]	●	◐	●	○	●
Mutz et al. [178]	●	●	○	●	●
Snow et al. [19]	●	○	○	○	●
Srivastava et al. [21]	●	○	○	●	●
Strackx et al. [171]	●	●	○	○	●
Sun et al. [20]	●	○	○	●	●
Wagner et al. [176]	●	●	○	●	○
Yap et al. [177]	●	●	○	●	●
Barabosch et al. [16]	●	●	●	●	●

TABLE 6.1: Matching of most related approaches with our five requirements to dynamically detect HBCIAs. Each requirement can be either fulfilled (●), partially fulfilled (◐) or unfulfilled (○).

We matched selected related work (see Chapter 3) to our five requirements in Table 6.1. This table shows that no candidate matched all five requirements. Therefore, we decided to close the gap and implemented *Bee Master*.

6.1.2 Overview

Bee Master transfers the honeypot paradigm to OS processes in order to detect HBCIAs. In a nutshell, it creates processes and observes them for signs of attacks. Since it knows the state of these child processes apriori, any deviation is considered suspicious. Examples for such deviations are new memory pages, new threads, or significantly modified memory pages within the processes. Therefore, we detect HBCIAs without the knowledge of any special OS API (e.g. debugging APIs on Windows). Our idea only relies on common concepts like processes, threads, and memory pages. For instance, Windows and Linux implement these concepts.

Figure 6.1 outlines *Bee Master*'s architecture. It consists of two components. The *Queen Bee* observes child processes for signs of HBCIAs. These processes are called *Worker Bees* and the *Queen Bee* spawns them as child processes. This ensures that it can introspect *Worker Bees*. Since every action within the *Worker Bees* is observable, the *Queen Bee* can detect HBCIAs in them.

We described HBCIA algorithms in Section 4.3. The first step is the victim process selection. The code-injecting malware selects its victim processes based on some features

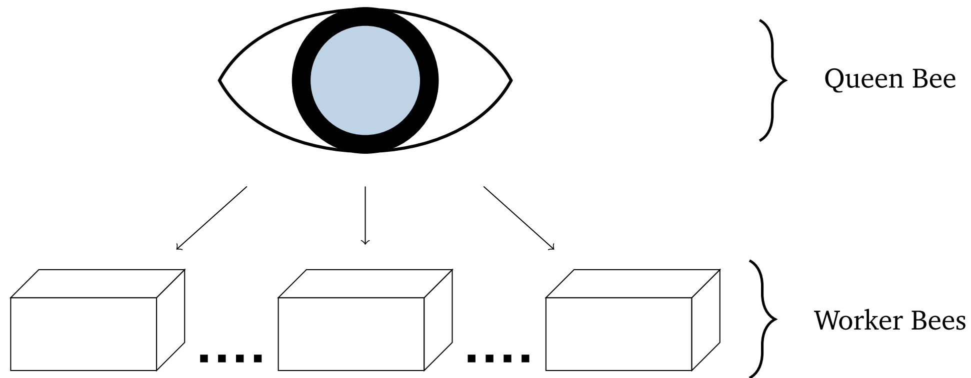


FIGURE 6.1: Overview of Bee Master's architecture: the *Queen Bee* and its *Worker Bees*.

(*Targeted Injection*), e.g. the process name, or it just tries to inject code into every accessible process (*Shotgun Injection*). To detect a *Shotgun Injection*, we have to deploy at least one process and observe it. To detect a *Targeted Injection*, the *Worker Bees* are configurable in order to resemble processes such as browsers or mail clients. Therefore, they trick malware into injecting code into them due to the fact that they exhibit the feature the malware looks for. As we will see later, there are several features that malware may utilize to identify its victim processes. To the best of our knowledge, we are not aware of any HBCIA-employing malware that checks the genuineness of a victim process.

6.1.3 Queen Bee

The *Queen Bee* is the main component of *Bee Master*. Its objective is to manage the *Worker Bees* and to monitor their internal states. *Worker Bees* mimic a ϵ_{victim} in order to lure HBCIA-employing malware to attack them. The *Queen Bee* continuously compares their states to the expected states that are apriori known. This includes the threads, the memory pages of the *Worker Bee*'s process including their content, and the loaded modules. If it detects a deviation then it detects an HBCIA. As a consequence, it creates a memory dump of the process and terminates it. The memory dump aids analysts in their further analysis of the caught code-injecting malware. If *Bee Master* is employed on a real system then the user should be warned and the system halted.

Figure 6.2 shows how the *Queen Bee* manages a *Worker Bees*. At first, it creates a *Worker Bee*. It depends on the privilege level of the actual implementation as to how this is achieved. A user space implementation relies on the APIs provided by the OS, a kernel module implementation directly creates these processes from kernel space, and a virtual machine introspection (VMI) component manipulates the OS from outside. We discuss the different implementation possibilities in Section 6.1.3.1. For now, let us suppose

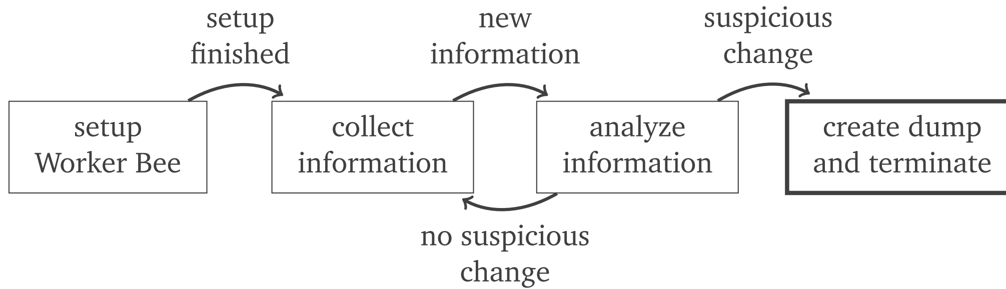


FIGURE 6.2: Control flow of the *Queen Bee's Worker Bee* handling algorithm.

that the *Queen Bee* is a user space process. Then it would call, for instance, the OS APIs *exec* or *CreateProcessA* on Linux or Windows, respectively. After the *Worker Bee* creation, the *Queen Bee* switches into a continuous monitor mode.

In this monitor mode, it first collects internal information about the *Worker Bee*. This includes the number of running threads, the loaded modules, and the memory pages. We stated in Section 4.3 that an HBCIA always needs memory to store the injection code (memory pages/modules) and an execution context to execute this code (threads). Hence, we collect fundamental information to detect HBCIAs. Again, there are several ways to achieve this depending on the implementation. As we suppose that our example runs in user space, this could be achieved via the API call *CreateToolhelp32Snapshot* on Windows. Next, it compares this information to the expected state of the *Worker Bee*, e.g. two expected threads versus three currently running threads. This is possible since these parameters are known apriori. If there is a deviation then this means that it detects an HBCIA. Hence, it breaks out of the monitor mode, creates a memory dump of the *Worker Bee* for further analysis, and halts it.

6.1.3.1 Decision Heuristic

Algorithm 1 shows the decision heuristic of *Bee Master* as pseudo code. The algorithm takes as input the set of currently running *Worker Bees* and outputs a set of HBCIA attacked *Worker Bees*. Note that the output set may be empty if no attack has occurred. The algorithm iterates over all *Worker Bees* and compares their apriori known state to the current state. If there is a deviation then it assumes a *Worker Bee* w_i to be attacked. The state comparison comprises four individual comparisons. The first three (lines 2,4 and 6 in Algorithm 1) are just comparisons of the set cardinality of the threads, memory pages, and modules of a *Worker Bee* w_i.

The fourth comparison (line 8 in Algorithm 1) compares the states of the memory pages of w_i in depth. To this end, we require a similarity measure that is robust to small

Input: $W := \{w_1, ..., w_n | w_i \text{ is running}\}$ where $n \in \mathbb{N}_{>0}, i \in \mathbb{N}_{>0} \land 1 \leq i \leq n$
Output: $A := \{w_i | w_i \in W \land w_i \text{ has been attacked}\}$

```
1:  for all w_i ∈ W do
2:      if |w_i.known_memory_pages| ≠ |w_i.current_memory_pages| then
3:          A ← A ⋃ {w_i}
4:      else if |w_i.known_modules| ≠ |w_i.current_modules| then
5:          A ← A ⋃ {w_i}
6:      else if |w_i.known_threads| ≠ |w_i.current_threads| then
7:          A ← A ⋃ {w_i}
8:      else if change(w_i.known_memory_pages, w_i.current_memory_pages) then
9:          A ← A ⋃ {w_i}
10:     end if
11: end for
12: return A
```

ALGORITHM 1: Pseudo code of the decision heuristic of *Bee Master* that the *Queen Bee* implements.

Input: Sets of memory pages $X := \{x_1, ..., x_k\}$ and $Y := \{y_1, ..., y_k\}$ where $k \in \mathbb{N}_{>0}$
Output: $\begin{cases} 1, & \exists SM(x_i) \neq SM(y_i) \\ 0, & \text{otherwise} \end{cases}$

```
1: for i = 1 to k do
2:     if SM(x_i) ≠ SM(y_i) then
3:         return 1
4:     end if
5: end for
6: return 0
```

ALGORITHM 2: Pseudo code of the memory page comparison algorithm of *Bee Master* that the *Queen Bee* implements.

changes in the memory pages. For now, we just assume that there is such a similarity measure SM that exhibits this property. We will explain the choice of SM later on. Note that if the algorithm reaches this point then there have not been any new pages and the cardinality of the sets $w_i.known_memory_pages$ and $w_i.current_memory_pages$ are equal. This fourth comparison is carried out by Algorithm 2. This algorithm takes as input two sets X and Y of memory pages and outputs 1 if there is at least one memory page that has significant changes or 0 otherwise. This algorithm iterates over all memory pages and computes the similarity measure SM of each corresponding pair in X and Y to compare them in order to detect significant changes that go beyond a couple of bytes. An example would be, for instance, the replacement of the main program image, which happens often in *Thread Manipulation* based HBCIA algorithms.

Choice of the Similarity Measure Algorithm 2 depends on a similarity measure SM. We have stated that we require a measure SM to be robust to minimal changes in the memory pages. Such changes may be caused by changes to variables in the data segment of a binary.

Strict hash sums of cryptographic hash algorithms like *MD5* or *SHA512* violate this property per definition. Their hash sum changes when a bit is flipped. Hence, they are not suitable for our SM. An alternative would be *Context Triggered Piecewise Hash* (CTPH) as suggested by Kornblum [60]. It identifies ordered homologous sequences even if one of the two compared inputs is a modified version of the other. Therefore, they allow us to have a flexible notion of equality. For instance, let us assume a binary with a data section comprising several variables that are adjusted during runtime. The source code of this binary does not change. We can still identify this binary as the same binary when using CTPH even though the data section has been modified. Therefore, CTPH is a reasonable choice that fulfills the robustness property mentioned above.

Runtime To estimate the runtime of *Bee Master*'s decision heuristic, we first have to estimate the runtime of Algorithm 2. Kornblum [60] stated that the cost to compute a $CTPH$ is $\mathcal{O}(b \log b)$, where b is the input size in bytes. We are required to compute the $CTPH$ twice, once for each of the two memory pages that we compare. A comparison of two $CTPHs$ is a constant operation of c. Hence, this yields $\mathcal{O}(2 \cdot (b \log b) + c)$, which can be simplified to $\mathcal{O}(b \log b)$. Given k memory pages in the sets of memory pages X and Y then this yields $k \cdot \mathcal{O}(b \log b) = \mathcal{O}(k \cdot b \log b)$.

Given the runtime of Algorithm 2, we can estimate the runtime of Algorithm 1. We assume that testing the inequality of the cardinality of two sets is a constant operation of c. Therefore, the algorithm yields a runtime of $\mathcal{O}(n \cdot ((k \cdot b \log b) + 3c))$. This equals to $\mathcal{O}(n \cdot k \cdot b \log b)$, where n is the number of *Worker Bees*, k is the number of memory pages to compare per *Worker Bee*, and b is the input size to compute a $CTPH$.

Privilege Level of the Queen Bee

In the previous section, we stated the implementation possibilities of the *Queen Bee*: a user space program, a kernel module, or a VMI component. In this section, we compare the different implementation possibilities of the *Queen Bee* and we discuss our implementation decision of the prototype.

The three different implementation possibilities user space, kernel space, or VMI component imply three different privilege levels: unprivileged, privileged, and hypervisor. Each privilege level comes with higher integrity since it is harder for malware to manipulate the implementation and with lesser detectability since the implementation runs with higher privileges than the malware. On the downside, the higher the privilege level, the higher the implementation cost. This is twofold: First, implementations in user space have access to a wealth of different API functions, which is not the case in kernel space and VMI. Therefore, we have to implement this functionality on our own.

Implementation	Privilege Level	Integrity	Detectability	Implementation Cost	Bare-Metal
user space	○	○	●	○	✓
kernel space	◐	◐	◐	◐	✓
VMI	●	●	○	●	✗

TABLE 6.2: Comparison of the three different privilege levels of the implementation. Each category has three relative degrees: low (○), medium (◐), and high (●). Furthermore, the table lists whether (✓) or not (✗) an implementation allows bare-metal deployment.

Second, implementations in user space can be debugged easier as they influence the system stability less than kernel space and VMI implementations. Table 6.2 summarizes our arguments for the three different privilege levels. The solution with the lowest privilege level and lowest integrity – user space implementation – has the lowest implementation cost. On the contrary, the VMI implementation has the highest privilege level and integrity but also the highest implementation cost.

We decided to implement a first proof of concept of *Bee Master* as user space program. First of all, for us as scientists, it is important to show that the concept works: Does malware fall into our trap of fake *processes*? The privilege level of our system's controller is secondary for corroborating this. Thus, we shunned the higher complexity of implementing the *Queen Bee* in kernel space or even as a VMI component in order to have a higher level of integrity at the expense of the higher implementation costs. Furthermore, the VMI implementation does not allow the system to directly run on bare-metal but rather it requires first a hypervisor. This would limit the portability idea of our system.

Our evaluation (see Section 6.2) shows that the prototype is very effective. The ongoing arms race between the malware authors and malware analysts may require an implementation at a higher privilege level. Therefore, we recommend implementing the *Queen Bee* with a higher privilege level to ensure its integrity in future implementations.

6.1.4 Worker Bees

Worker Bees are the second component of *Bee Master*. A *Worker Bee* can be thought of as a sensor that allows the detection of HBCIAs. They are regular OS processes that mimic ϵ_{victim} processes. The *Queen Bee* handles one or more of them simultaneously. This increases the chances that an ϵ_{inject} attacks them. They behave passively, idling until a compromise happens. The user can configure the *Worker Bees* to mimic important aspects of real processes like a browser. For instance, since many malware families identify their victims via the process name, this is an important aspect to them. Hence, the user can configure, for example, the process name to match the needs of malware.

Parameters of the Worker Bees

At the moment, there are six parameters. We chose these parameters based on our domain knowledge in the field of malware analysis and our experience with HBCIA-employing malware in particular. In the following, we present them and discuss our choice.

(I) **threads**: Threads are an important building block to offer the user a multi-tasking experience. Single-threaded applications may raise suspicion, hence the analyst can adjust the number of threads.

(II) **loaded libraries**: System libraries offer a wide range of functionality to a program including networking and cryptography. The loaded libraries expose information about the purpose of a process, e.g. a browser requires libraries to encrypt network traffic.

(III) **memory mapped files**: Memory mapped files are a concept that is present on several operating systems including Microsoft Windows and Linux. A memory mapped file correlates, for example, with a full or partial mapping of a file from hard disk, a shared memory object between processes, or a hardware device [260]. This allows developers to manipulate file objects like regular memory. Malware identifies processes based on their file mappings.

(IV) **process name**: Often malware identifies interesting processes via the process name, e.g. banking Trojans like *Zeus* search for process names such as *iexplore.exe* or *chrome.exe* [38].

(V) **process window name**: Another way to identify interesting programs is the process window name, e.g. the malware *Gapz* searches for a window with the name *Shell_TrayWnd*, which belongs to *explorer.exe* [232].

(VI) **command line string of the process**: The command line string of a process reveals a lot about the process' purpose. Malware identifies interesting processes via the command line string, e.g. a networking service like *svchost* on Windows with the command line string *svchost.exe -k netsvcs* [46].

Please note that we intended *Bee Master* to be enhanceable and thus new parameters can be added. A possible scenario would be that a malware family identifies its target process via a file handle that it holds. Then, the *Worker Bees* could be enhanced to hold file handles and hence trick the malware into attacking them.

6.1.5 Implementation

We implemented *Bee Master* for Windows as well as Ubuntu Linux. The first proof of concept was implemented as a user space program as discussed in Section 6.1.3.1. The Windows implementation utilizes the Windows Debugging API [54]. While this allows a clean implementation of the idea, malware may detect that another process debugs it. However, this shortcoming does not apply to the concept of *Bee Master*. The Linux implementation utilizes *ProcFs* [261] to gather information.

6.1.6 Limitations and Evasions

We discuss limitations and evasions of *Bee Master* in this section. At first, we elaborate on the limitation of process hollowing detection and limited process coverage. Then, we discuss possible ways to evade our system.

Limitation: Detection of Process Hollowing

Bee Master does not detect *Process Hollowing*, which is a variant of *Thread Manipulation* (see Section 4.3), where the attacker requires full control over the victim process. Therefore, ϵ_{inject} usually creates ϵ_{victim}. Hence, *Bee Master* is not capable of detecting such HBCIAs since the *Queen Bee* can not observe processes that it has not created. However, we showed in Chapter 5.4 that *Concurrent Execution* is more common than *Thread Manipulation* of which *Process Hollowing* is a subset. Credential stealers still have to inject code into, for instance, browsers that have not been started by them. Hence, *Bee Master* should still cover a great share of code-injecting malware.

Limitation: Limited Process Coverage

The detection of HBCIAs depends on the process identification feature that the malware employs. Such features are, for instance, the process name, the process window name, and memory mapped files. It is impossible to provide all possible feature combinations. As a consequence, our system may miss attacks because the malware may not identify one of our *Worker Bees* as its victim process. However, two facts mitigate this problem. First, if the malware utilizes *Shutgun Injections* then the aforementioned is irrelevant. Second, we determined that there are several victim processes that malware prefers (see Section 5.2). Our default configuration covers these prevalent victim processes and hence increases the chance of an attack.

Discussion of Evasion

As every detection system, *Bee Master* may be evaded provided that the attacker has sufficient knowledge of its internals and is willing to deviate from standard techniques that are established in malware programming. An attacker would have to identify that a *Worker Bee* is not genuine. This would require an attacker to access the process of the *Worker Bee* and compare its internals to the expected internals of a certain process, e.g. loaded libraries or running threads. At the time of writing, we have not seen this behavior in practice. If malware accesses a process, for instance, with *OpenProcess*, then this is detectable. Furthermore, we designed our system to be enhanceable and configurable to be successful in the ongoing arms race between attackers and defenders.

6.2 Evaluation

We evaluate the prototype implementation of *Bee Master* in this section. Since our approach is not limited to a certain operating system and the prototype implementation runs on Windows and Linux, we evaluated it on both systems to show its platform-independence. However, the main focus is on Windows owing to its prevalence as a malware target, especially in the case of HBCIAs. Therefore, we were able to acquire more malicious samples on Windows for the evaluation.

First, we describe the evaluation methodology. Next, we describe the configuration of *Bee Master* that we employed throughout the evaluation and present the data sets for Windows and Linux. Subsequently, we proceed to evaluate *Bee Master*'s ability to detect HBCIAs on Windows and Linux. In this evaluation, we show that it handles a broad variety of prevalent malware families as well as artificially but potentially novel code injections on Linux.

6.2.1 Evaluation Methodology

Bee Master is a dynamic analysis system. Therefore, we had to execute the malware samples in order to evaluate how well our system detects their HBCIAs.

We conducted this evaluation as shown in Figure 6.3. At first, we prepared a VM with our implementation running and took a snapshot of this clean state. We configured *Bee Master* as described in Section 6.2.2. Then, for each sample, we restored the snapshot of the VM ① and executed it in this VM ②. After five minutes, we extracted the logs as well as dumped files from the VM ③ and reverted the VM to its original state ④. Note

that two to five minutes are common timeouts in sandboxing systems, which is based on domain knowledge. To the best of our knowledge, there is no evaluation of this parameter.

To show *Bee Master*'s platform agnosticism, we evaluated it on Windows and Linux. We chose three Windows versions XP, 7, and 8. Windows XP was chosen because it is the classical malware target on which many malware families work since the security measures are limited. We chose Windows 7 because it is still the Windows version with the greatest market share [8] (as of November 2017) and Windows 8 because it was the latest Windows version at time of the evaluation. All Windows systems are x86 systems because in our experience the majority of malware families still focuses on this architecture. In the case of Linux, we chose Ubuntu Linux 13.04 because it is the version that the only HBCIA-employing malware in our corpus demands (see Section 6.2.3) and Ubuntu 17.10 because it is the latest version. Both Linux systems are x64 systems because this is the default in the Linux Desktop universe.

To summarize, we evaluated *Bee Master* on the following operating systems:

- Windows XP SP3 32 bit
- Windows 7 SP1 32 bit
- Windows 8 SP0 32 bit
- Ubuntu Linux 13.04 64 bit
- Ubuntu Linux 17.10 64 bit

The hypervisor was VirtualBox 4.2.10 [72]. We installed all VMs without additional software and we hardened them against several VM detection methods to cope with evasive malware [47]. None of the VMs was connected to the Internet (see Section 5.3). Furthermore, we did not simulate user interaction.

6.2.2 Configuration of the Prototype

This section describes how we configured *Bee Master* for the evaluation. We utilized the default configuration during the evaluation. The default configuration differs for Windows and Linux. We created them based on our experience with HBCIA-employing malware but we also included the top processes of the evaluation of the victim prevalence (see Section 5.2). Hence, the Windows configuration composed the following five ϵ_{victim} processes:

Data Set	R1	R2	R3	R4
cwsandbox	✗	✗	✓	✗
Android Malware Genome Project	✗	✗	✗	✗
Malicia	✗	✗	✓	✗
Drebin	✗	✗	✗	✗
Microsoft Malware Classification Challenge	✓	✗	✓	✗
Android Malware Dataset	✗	✓	✗	✗

TABLE 6.3: Matching of publicly available data sets to our requirements of Section 6.2.3.

- the Windows shell (explorer.exe)
- the default Microsoft Internet browser (iexplore.exe)
- an alternate browser (firefox.exe)
- a networking service (svchost.exe)
- a random process (pdtyzgxm.exe)

The processes *explorer.exe*, *iexplore.exe*, and *svchost.exe* are all preferred victims (see Section 5.2). We chose *firefox.exe* because it is a popular browser that banking Trojans target and a random process name to discover malware families that employ a *Shotgun Injection*. The Linux configuration just comprised two ϵ_{victim} processes:

- a popular browser (firefox)
- a random process (pdtyzgxm).

These two victims were chosen for the same reasons as in the Windows case.

6.2.3 Data Sets

The following two sections describe the data sets employed in the Windows and Linux evaluation of *Bee Master*.

Data Set Windows

Our objective was to evaluate a dynamic HBCIA detection system that should run in Windows environments with goodware and possibly malware. Unfortunately, related

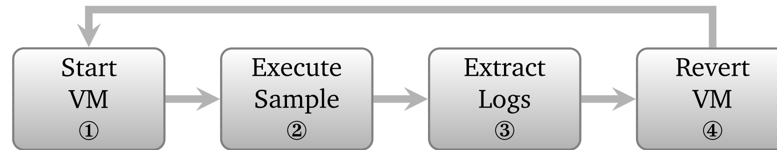


FIGURE 6.3: Methodology of the evaluation of *Bee Master*

publications did neither publish their evaluation data sets nor did they state requirements for an evaluation data set (see Chapter 3). Therefore, we formulated our own requirements.

Since we wished to evaluate an HBCIA detection system, we required HBCIA-employing malware families. In Section 6.1.6, we stated that one limitation of our system is the detection of *Process Hollowing*. Therefore, we refrained from evaluating HBCIA-employing malware that utilized this technique. As a result, we required the data set to include HBCIA-employing families without families that employed *Process Hollowing* (**R1**).

There are millions of known malware samples [4]. However, many samples of them are not longer relevant (e.g. DOS boot sector viruses). Hence, we required samples that were still found in the wild. Furthermore, there are many utilization schemes of malware (see Section 2.1). A data set with only banking trojans would be biased. Therefore, a wide-variety of malware classes should be included to adequately represent the threat landscape. As a consequence, **R2** required a recent and a wide-variety of malware families in the data set.

Even though in a time of operation system diversity, Microsoft Windows is still the market leader in Desktop operating systems [8] (as of November 2017). Therefore, it is the preferred target of malware authors. The probability that the information they are looking for is handled by a Windows system is high. Malware detection systems should primarily focus on Windows but without losing secondary focus from the other possible malware targets. Consequently, the malware should be for the most prevalent malware target Windows (**R3**).

Finally, it is not sufficient to measure the detection rate in terms of true positives/false negatives. There should also be goodware programs run against our system in order to measure false positives/true negatives. The goodware programs should be representative. Consequently, **R4** stated that the data set had to contain representative goodware programs.

To summarize, we had the following requirements for a data set for evaluating a dynamic HBCIA detection system:

R1 HBCIA-employing malware families (excluding *Process Hollowing*)

Operating System	Malware Families	Not Working Families
Windows XP	38	-
Windows 7	37	*Poison*
Windows 8	32	*Bamital, Conficker, Gamker, Ice X, Poison, Sykipot*

TABLE 6.4: The number of working malware families and the families that refuse to work on the Windows versions.

R2 recent and a wide-variety of malware families to represent the threat landscape

R3 only malware for the most prevalent malware target Windows

R4 goodware programs to estimate false positives

Table 6.3 matches our four requirements to publicly available malware data sets (see Section 5.1). This table shows that none of them satisfies our requirements. Therefore, we opted to create our own data set.

We compiled the data set as follows: First, we added 38 representatives of known HBCIA-employing malware families that we had gathered throughout the last years. We only added one representative per family since HBCIAs are an inherent family feature (see Section 5.3) and hence one family member is sufficient. The code injecting capabilities of the malware families were manually verified in all cases. We did not add families that utilize *Process Hollowing* since *Bee Master* does not detect them (**R1**).

All samples ran on Windows XP but not all of them on Windows 7 and Windows 8. Reasons for this are manifold. For instance, the malware is not compatible with newer Windows versions. Two reasons are the incompatibility between the WinAPI versions (especially when using low-level API calls) and the advancement in exploit mitigation techniques introduced by the newer Windows 7 and 8. Table 6.4 lists the number of working families and points out which families did not execute on Windows 7 and Windows 8. All families are listed in Appendix A.

To fulfill requirement **R4**, we added goodware to estimate false positives. We collected the goodware from Windows system tools and added popular portable apps like browsers, instant messaging clients, and encryption software. Table 6.5 lists the data per Windows version.

Data Set Linux

Code injections are prevalent on Windows (see Section 5.1). Even though they can be encountered on other platforms such as Linux [10], macOS [11], and Android [210]

Operating System	Windows System Tools	Portable Apps	Total
Windows XP	321	13	334
Windows 7	440	13	453
Windows 8	470	13	483

TABLE 6.5: Summary of the Windows goodware per Windows version. The data set comprises Windows system tools and popular portable apps.

as well, they are (not yet) prevalent on them. In addition to evaluating *Bee Master* on Windows, we opted to evaluate it on Linux to show its platform-independence. We chose Linux due to its openness, easy access, and good documentation of platform internals.

To the best of our knowledge, there was no publicly available data set that could have been used to evaluate an HBCIA detection system on Linux. Also, we could not utilize the data set that we built before since they do not comprise Linux software. Therefore, we built another corpus to evaluate our system on Linux.

Whereas gathering benign programs on Linux was easier than on Windows due to the package management systems of the various Linux distributions, gathering malicious programs that utilized code injections was difficult. We gathered 1425 benign programs by searching the filesystem tree of */usr/bin* for *ELF* executables. The system was an actively used developer system with office, networking, developer, and multimedia tools installed. A list of the program hashes can be found at [262].

Unfortunately, we encountered only one Linux malware family that utilizes HBCIAs (the banking Trojan *Hanthie* [10]). Linux is not a popular malware target like Windows. There are many different Linux distributions that do not adhere 100% to standards. Furthermore, the user base of them is much smaller and also more technical, maybe resulting in a higher cyber security awareness. Nevertheless, Linux is prone to code injections. There are proof of concept HBCIA implementations for Linux such as *linux-inject* [263] and *linux-injector* [264].

To evaluate *Bee Master* on Linux, we decided to implement several proof of concepts of HBCIA techniques. We called the resulting framework *1001-injects* and released its source code on github [265] to allow others to study HBCIAs on Linux. Some injections are based on previous work like *linux-inject* [263] or *linux-injector* [264]. However, some of them are new and to the best of our knowledge they have never been publicly mentioned. Table 6.6 summarizes the injection methods of *1001-injects*.

To conclude, we gathered 1431 samples in total. Whereas 1425 were benign programs, six were HBCIA code injection techniques including the malware family *Hanthie*.

Technique	Previously Implemented
inject ELF library dlopen	*linux-inject* [263]
inject shellcode hijack thread	unknown
inject shellcode new thread	unknown
inject shellcode new pthread	unknown
inject ELF executable via /dev/mem	unknown

TABLE 6.6: *1001-injects*: summary of injection techniques for Linux. The proof of concept implementation is hosted on github [265].

6.2.4 Results

We present the results of the Windows and Linux evaluation in the following two sections.

Results Windows

First of it all, *Bee Master* detected the code injections in all cases on all three Windows versions. The malware attacked at least one *Worker Bee*. The *Worker Bees* mimicked preferred victim processes, hence this came as no surprise. This means that our system detected the malicious behavior in all cases on all three Windows versions. Furthermore, our system did not falsely detect code injections during the execution of goodware. All goodware samples that we executed with *Bee Master* on the three Windows versions did not exhibit any sign of an HBCIA.

Table 6.7 lists the code injections per *Worker Bee* for the malicious samples on Windows XP, Windows 7, and Windows 8. Most families injected code into *explorer.exe*. For instance, on Windows XP 34 families targeted this process, which accounted for 89%. This process was followed by *iexplore.exe*, also a default Windows program. We observed the least injections into *firefox.exe*. The reason for this may be that mostly banking Trojans target this browser, for example, to intercept credentials. Therefore, an injection into this browser may indicate the presence of a banking Trojan or a malware that utilizes *Shotgun Injections*. Furthermore, many families attacked the random process. On all three Windows versions, they accounted for more than 50% of the data set. However, not all of them utilized *Shotgun Injections*.

Table 6.8 lists how many families attacked which number of *Worker Bees* on Windows XP. It shows that 42% of the families targeted all five of them including the one with the random process name. This suggests that they utilized *Shotgun Injections*. However, it is not decidable from *Bee Master*'s point of view whether or not a malware family utilizes this kind of injection since it does not have a global view of all processes. Another

Process	Windows XP	Windows 7	Windows 8
explorer.exe	34	27	24
iexplore.exe	28	29	22
svchost.exe	26	21	20
firefox.exe	24	21	23
pdtyzgxm.exe	28	23	20

TABLE 6.7: Summary of the observed injections into the *Worker Bees* employed on Windows XP, 7, and 8.

21% targeted four *Worker Bees*. Families that attacked more than four processes include banking Trojans and credential stealers such as *Cridex*, *Hesperbot*, and *Zeus*. They are not interested in hiding but rather in intercepting information. Another insight is that many families attacked the *Worker Bee* with the random process name but shunned another process. This suggests that these families employed *Targeted Injections* via a black list to exclude system processes that they did not want to attack.

The evaluation also shows that it may not be required to employ many *Worker Bees* since all malicious samples of the data set attacked at least either the Windows shell *explorer.exe*, which always runs or the browser *iexplore.exe*, which is a default Windows program. As seen in Section 5.2, both are highly ranked victim processes.

In conclusion, this Windows evaluation showed that *Bee Master* detected a wide variety of malware families at an early stage of their infection process and without misclassifying any goodware. It also confirmed the finding of Section 5.2 that stated the absolute preference of the process *explorer.exe*. We saw that more than one-third attacked five *Worker Bees* suggesting that they utilized *Shotgun Injections*. Note that this also included the *Worker Bee* with the random process name that malware authors could not anticipate. Hence, we assume that this victim was not specifically selected. The key observation is that the whole data set could have been covered with only two *Worker Bees*: *explorer.exe* and *iexplore.exe*.

Results Linux

The results of the Linux evaluation are in line with the Windows evaluation. *Bee Master* detected all six injection types. One of them was a banking Trojan found in the wild and five were artificial code injection techniques. There were no false positives due to the execution of the 1425 benign samples.

Even though code injections are not as prevelant on Linux-based systems as on Windows systems, *Bee Master* is already capable of detecting several distinctive code injection

Number of Attacked *Worker Bees*	Number of Families	Total Percentage
1	5	13.2%
2	4	10.5%
3	5	13.2%
4	8	21.1%
5	16	42.1%

TABLE 6.8: Distribution of families to the amount of attacked processes on Windows XP.

techniques. Some of them are not even found in the wild. Furthermore, this evaluation has shown that *Bee Master* is a promising step towards operating system agnostic *HBCIA* detection, since we utilized it to detect these attacks on Linux-based platforms.

6.3 Conclusion

We have presented a dynamic HBCIA detection system. *Bee Master* applies the honeypot paradigm to OS processes to detect HBCIAs. It consists of two components: The *Queen Bee* and its *Worker Bees*. The *Queen Bee* continuously checks all its *Worker Bees*. The internal states of the *Worker Bees* are apriori known. This includes the number of running threads and loaded libraries. Therefore, the *Queen Bee* detects suspicious deviations within a *Worker Bee* since HBCIAs affect these parameters, e.g. injected code loads additional libraries. Once it has detected an HBCIA, it creates a memory dump of it for further analysis. After successful dumping of the memory, it terminates its *Worker Bee*. *Worker Bees* are configurable to mimic potential victims like browsers.

There are several advantages of our approach when compared to related work. It runs on commodity hardware and does not require special hardware. It can be implemented in user space, kernel space, or as VMI module and it does not require any modifications of its target system such as Windows and Linux. There are no special APIs that *Bee Master* requires in order to detect HBCIAs. It just needs knowledge of threads, modules, and memory pages in its child processes, which are common concepts in modern operating systems. All these advantages yield a portable approach that is required by today's diverse operating systems.

We implemented a prototype of *Bee Master* that works on Windows and Ubuntu Linux. In an evaluation on Windows and Ubuntu Linux, we have shown that it reliably detected code injections and it did not exhibit false positives. On the one side, we showed that malware does not verify if its ϵ_{victim} is genuine. Current malware is indeed very vulnerable to detection at this stage of its execution. On the other side, we expected that there

would not be any false positives since it seems very unlikely that end user programs need to access the process space of other processes or start further threads within them.

Donde fueres, haz lo que vieres

refrán español

7

Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps

Our approach *Bee Master* that we have presented in the previous chapter dynamically detects HBCIAs. It prevents greater damage by reporting these attacks to the user at runtime. In this chapter, we focus on a different scenario: static detection of HBCIAs in memory dumps. The memory dumps come from victim machines that were compromised and whose data had already been breached. The goal in this scenario is to detect the malware that was responsible for the attack. It can then be analyzed to get an impression of the possible damage, such as data theft, data manipulation, or data encryption. Furthermore, it may be possible to attribute the attack to a certain threat actor, though this is a difficult challenge [256].

Forensic and malware analysts encounter HBCIAs daily. Both employ forensic memory analysis. There have been notable advances in forensic memory analysis due to open source frameworks like *Volatility* [25]. This framework closes the semantic gap between the binary data in memory and the meaning of this data to the operating system. In a nutshell, it finds the relevant data structures, e.g. pointed to by special registers, and gradually resolves further links to other data structures until the semantic gap is closed. By doing that, it recovers the meaning of this data and allows the user, for instance, to examine the running processes or loaded kernel modules.

Forensic analysts analyze memory dumps of unknown systems without any knowledge about the intrusion technique used. The detection of the injected code is a first step towards the solution of a case. Also, malware analysts incorporate more forensic analysis techniques in their work flow due to the advances in the field of memory forensics. In contrast to forensic analysts, they execute the malware in a known environment. They profit from the fact that memory forensics gives them a bird eye's view into their analysis environment. Since the analysis does not happen on the infected system, the malware can not tamper with the results and forge, for example, the list of running processes to hide rootkit components. Even though the initial situation of both analyst groups is different, it is essential to both to quickly find the injected code in order to continue the (malware) analysis.

Current memory based HBCIA detection systems like *Malfind* [25], *Hollowfind* [27], and *Membrane* [28] suffer from severe drawbacks. The current state of the art *Malfind* exhibits a high false positive rate since it relies only on two features to detect HBCIAs. The *Volatility* plugin *Hollowfind* [27] detects only a fraction of all HBCIAs because it focuses on *Process Hollowing*. Both fail to detect widespread malware families like *Dridex* that inject a dynamic linked library (DLL). The academic approach *Membrane* is restricted to a coarse grain detection. In contrast to the aforementioned systems, *Membrane* does not detect injected memory regions but victim processes. This is not sufficient since victim processes can contain hundreds of memory regions to search in. For example, the Windows process *Explorer* has more than 550 memory regions that account for more than 400 MB of data on an idling Windows 10 system.

In this chapter, we present *Quincy* designed to overcome the drawbacks of current systems. At its heart, it employs a machine learning heuristic based on up to 36 features associated with HBCIAs. These features include, for instance, the presence of shellcode, the protection of a memory region, or the memory region compression ratio. It employs a tree-based machine learning algorithm, which can handle non-linearity. We discard less valuable features by means of a feature selection for each evaluated operating system. Besides lowering the analysis speed, it improves also the detection performance by disposing of suboptimal inter-feature relationships. *Quincy*'s detection heuristic is as fine grained as *Malfind's* and *Hollowfind's*. Hence, it is finer than *Membrane's* detection heuristic. Furthermore, it raises the bar for an attacker to circumvent this approach due to its up to 36 different features, which work in conjunction.

Our evaluation of *Quincy* shows that it improves upon the state of the art *Malfind* and Hollowfind. We created a high quality data set with a sound ground truth for three Windows version including the latest Windows 10. On the one side, our system significantly lowers the false positive rate when compared to the other contenders. On the other

side, *Quincy* improves upon their true positive rate, detecting more advanced HBCIAs like injection of DLLs, which exhibit proper memory permissions.

The remainder of this chapter is structured as follows: First, we introduce *Quincy* by discussing requirements for systems that forensically detect HBCIAs in memory dumps. Next, we introduce our approach in detail. This is followed by the evaluation of *Quincy* as well as two other detection systems. Finally, we conclude this chapter.

We pointed out in Chapter 4 that HBCIAs are a cross-platform problem. Furthermore, we showed in Chapter 6 that generic HBCIA detection is possible. Unfortunately, there are only few cases on non-Windows platforms. As a consequence, we created artificial cases in the evaluation of Chapter 6. Though, we believe that there will be more cases in the future. Therefore, we decided to implement *Quincy* only for Windows in a first step, since the number of available samples, for instance, for Linux are not sufficient to train and validate a machine learning model. Hence, we leave this to future work (see Section 8.2.3). Nevertheless, our system should be easily portable due to the underlying memory forensic abstractions introduced by the framework *Volatility* (see *Implementation* in Section 7.2.5).

Please note that partial results of the presented chapter were published in the seminal paper "Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps" at the conference "Detection of Intrusions and Malware, and Vulnerability Assessment" (DIMVA) in 2017 [17]. The proceedings were published by *Springer International Publishing*.

7.1 Requirements

The *Volatility* plugin *Malfind* proposed by Hale Ligh [25] is the current state of the art when it comes to detecting HBCIAs in memory dumps. *Malfind* implements a combination of features designed to decide whether a memory region is benign or malicious. First, it marks entirely empty memory regions as benign. Pages with *RWX* protections and unlinked libraries (from the *PEB*) are marked as malicious. Furthermore, *Malfind* detects wiped *PE* headers in *RWX*-protected memory regions. Finally, all unflagged regions are labeled as benign. *Malfind* has two drawbacks. First, it only detects the low-hanging fruits, i.e. simple code injections that leave obvious traces like memory regions allocated with *RWX* permissions. Second, it assumes too many benign memory regions to be malicious and hence it has a high false positive rate. This increases the workload of the analyst, who has to manually analyze these memory regions to confirm that they are benign. Monappa proposed *Hollowfind* [27] that detects a subclass of all HBCIAs. It

Approach	R1	R2	R3	R4
Malfind [25]	●	○	○	●
Membrane[28]	●	◐	◐	○
Hollowfind [27]	◐	○	○	●
HashTest [26]	●	○	◐	●
Barabosch et al. [17]	●	●	●	●

TABLE 7.1: Matching of related approaches to our four requirements to statically detect HBCIAs. Each requirement can be either fulfilled (●), partially fulfilled (◐), or unfulfilled (○).

focuses on *Process Hollowing*, a technique to hijack benign processes (see Section 4.3.3). Its detection heuristic is based on two features, which were engineered utilizing domain knowledge (see Section 3.3.1). There are two more proposals by Pek et al. (*Membrane*) [28] and White et al. (*Hashtest*) [26]. However, *Hashtest* utilizes a whitelisting approach and hence does not work with previously unseen systems. *Membrane* is neither capable of detecting memory regions (it just detects infected processes) nor publicly available.

As we have pointed out, available systems exhibit various drawbacks and we have identified several improvements. Unfortunately, related publications did not define any requirements for a system that statically detects HBCIAs in memory dumps. Therefore, we first define the requirements for such a system.

We wish to improve upon current HBCIA detection systems, which statically detect such injections in memory dumps. Therefore, we require a system with this objective (**R1**).

As we have already stated above, current approaches (e.g. [25, 27]) exhibit insufficient detection rates. They have many false positives, which cost analysts a lot of time to discard and they tend to focus only on specific approaches (e.g. *RWX* memory pages in *Malfind* or *Process Hollowing* in *Hollowfind*). As a consequence, the approach should improve upon current approaches. It should be more general to detect more malware using other injection techniques, but also it should lower the false positive rate to optimize the time an analysts has to study the results of our approach (**R2**).

The malware industry and the computer security industry play a cats and mouse game. Once a new detection system is published, the other side tries to bypass it. Current systems can be easily bypassed (see Section 7.2.6). Therefore, we require the system to significantly raise the bar for evasion (**R3**).

A Windows 10 runs dozens of processes with hundreds of memory regions within each of them. For example, the Windows process *Explorer* has more than 550 memory regions that account for more than 400 MB of data on an idling Windows 10 system. Hence, it is

very helpful if the system directly points the analyst to the infected memory region. As a consequence, **R4** states that the system requires a fine detection granularity on memory region basis.

To summarize, our requirements are:

R1 Detection of *Host-Based Code Injection Attacks* in memory dumps

R2 Improvement of the true positive rate/false positive rate, when compared to the state of the art

R3 Harder circumvention, when compared to the state of the art

R4 Fine detection granularity on memory region basis

Table 7.1 matches our requirements to related work. None of them matched all requirements. As a consequence, we developed our solution to satisfy them all.

7.2 Methodology

This section presents *Quincy*, our method to forensically detect HBCIAs in memory dumps. We provide an overview of the system and its phases: feature extraction, feature selection, learning, and classification.

7.2.1 Overview

We begin by providing a superficial overview of our system before we dive into its details. Figure 7.1 outlines the key points of *Quincy*.

First, it takes as input a memory dump and closes the semantic gap. The difference of what an operating system stores in the raw memory, i.e. a binary string, and what it interprets into this memory is denoted as *semantic gap* [70]. Internally, *Quincy* relies on the memory forensic framework *Volatility* [25], which detects and recovers key data structures in memory dumps to recover the original meaning (see also Section 2.2.3).

① Then, *Quincy* enumerates all memory regions of all processes of the memory dump and extracts features from them, e.g. the memory region protection, the presence of English strings, or the presence of code. In the following, we define a *memory region* as a consecutive set of memory pages within the boundaries of an arbitrary virtual process space. Whereas a memory page on a x86 system is typically four kilobytes [266], a

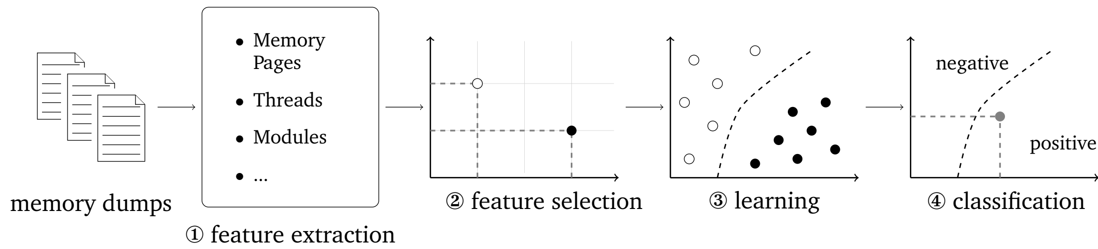


FIGURE 7.1: *Quincy* receives memory dumps with labeled memory regions as input. ① Then, it extracts 36 HBCIA-related features from seven categories. ② Subsequently, it discards invaluable features and embeds the rest in a vector space. ③ Next, it induces a binary classifier. ④ Finally, it classifies unseen memory areas.

memory region is a multiple of that and can easily be several megabytes. For example, on Windows a memory region is equivalent to a *Virtual Address Descriptor* (VAD) [266]. Furthermore, regions may be shared between processes. For instance, Windows maps system libraries with *EXECUTE_WRITECOPY* permissions. This shares the data among several processes to save memory space. Only when the process modifies the data, does the OS create a local copy of the data in the process and this copy is then modified [266].

② Subsequently, we conduct a feature selection. While this improves the detection performance due to the removal of weak features or linear dependencies, it also decreases the runtime since it reduces the problem dimensionality. We employ a recursive feature selection, which recursively reduces the set feature-wise until it encounters the optimal feature set.

③ Thereafter, *Quincy* trains machine learning models based on the reduced feature set. It employs several algorithms and selects the best performing model as final model. Internally, it utilizes the machine learning library *scikit-learn* [267].

④ Finally, *Quincy* classifies previously unseen memory regions as either benign or malicious, using the aforementioned optimized machine learning model. *Extremely Randomized Trees* work especially well as will be shown in the evaluation.

7.2.2 Feature Extraction

This section describes *Quincy*'s 36 features in detail. We have organized them in seven categories. The majority of the features is binary. However, there are also continuous features. Not all categories apply to every memory region. For instance, the category *binary* only applies to memory regions containing binary executables. If this is not the case then the features yield zero values.

Overview of *Quincy*'s features			
Category	**Feature**	**Rank**	**Description**
① **API**	dynamic_loading	25/23/17	presence of dynamic loading APIs
	general_api_strings	10/10/20	common API call prefixes
	hashing	01/04/06	code fragments related to API hashing
	hbcia_api_strings	07/02/02	common HBCIA APIs
② **binary**	exports	26/30/24	exports API calls
	has_header	20/15/19	starts with a header
	imports	29/27/11	imports API calls
	is_dynamic_library	23/21/22	has been loaded dynamically
	is_module	12/07/16	registered module known to the OS
	is_pe_or_dll	16/12/21	a PE executable or shared library
	wiped_header	35/34/32	executable header has been wiped
③ **code**	functions	18/14/10	common assembler function prologues
	hooks	08/09/23	memory region contains code hooks
	shellcode	03/11/13	shellcode patterns
④ **cryptography**	cipher	32/28/27	constants of ciphers
	encoding	21/18/09	constants of encoding schemes
	hashing	27/16/15	constants of hashing algorithms
⑤ **countermeasure detection**	debugger	24/26/31	strings and code patterns to detect debuggers
	sandbox	19/31/14	strings and code patterns to detect sandboxes
	vm	33/33/35	strings and code patterns to detect virtual machines
⑥ **memory**	embedded_exe	36/36/36	embedded executable after header
	english_strings	28/24/26	strings of Google's top 1000 English search terms
	high_entropy_areas	04/03/04	areas of high entropy
	is_heap	31/25/28	memory region is a heap
	is_sparse	05/01/03	ratio of zero bytes
	mapped	06/35/33	corresponds to a memory mapped file
	network_strings	02/05/18	strings related to networking
	persistence	22/19/05	strings related to persistence
	private	13/08/08	tagged as private memory
	protection	09/17/01	protection of memory region
	tag	14/13/07	tagged by allocation functions
	threads	15/06/12	threads originated in memory region
	victim_strings	11/29/25	names of typical HBCIA victims
⑦ **trojan**	banking	34/22/30	strings related to online banking
	cookies	30/32/34	strings related to cookie stealing
	credentials	17/20/29	strings related to credential stealing

TABLE 7.2: Overview of *Quincy*'s 36 features in categorical and alphabetical order. The features are ranked based on the *Recursive Feature Selection* (see Section 7.2.3). Quincy's final models may not utilize all features (see Appendix C).

Note that the initial version of *Quincy* utilized 38 features [17]. However, the features $code_{indirect_calls}$ and $code_{indirect_jumps}$ turned out to be rather harmful, even though they had high rankings in the feature selection. As a consequence, we have decided to manually remove these features in order to improve performance. This shows again that machine learning is still a ”black art” as stated by Domingos [268], which involves a lot of manual tweaking.

① **API** Malware is required to communicate with its environment. Therefore, it asks the OS to carry out actions like file access or network communication. Typically, the OS

provides system calls for these tasks. However, Windows system libraries like *kernel32* or *ntdll* abstract from low-level system calls and provide a programmer-friendly interface, known as the *Windows Application Programming Interface* (WinAPI or just API in the following). Internally, such APIs carry out system calls if needed. This shows how important API calls are to malware. Given the set of API calls of a malware, an analyst can quickly analyze the malware. Sometimes by just analyzing the sequence of the API calls in the malicious binary. There are several publications that rely on this method (e.g. [52, 269, 270]).

The prefix of an API call gives its type of action away. For instance, consider the API *CreateFile*. The prefix *create* tells us that this API call, which works on a *file*, actually creates such an object. There are many common prefixes including *create*, *get*, *open*, and *set*. Searching for them allows us to find general strings of API calls that programs utilize to communicate with the OS. Therefore, the feature $API_{general_api_strings}$ scans all memory regions for common API prefixes.

Since the presence of API calls aids the malware analysis process, malware authors seek ways to obfuscate the API usage to complicate the analysis. Current OSes offer functionality to load libraries at runtime. For instance, programmers can dynamically load libraries on Windows with the two API calls *LoadLibrary* and *GetProcAddress*. The feature $API_{dynamic_loading}$ searches for strings of functions that dynamically load libraries and APIs.

Advanced malware goes one step further and resolves pointers to API calls on its own. For this sake, it enumerates all system libraries that the process space maps. For each of them, it parses their exported API calls, hashes them, and compares them to an internal list of hashes, which is also called *API hashing* [271]. This eliminates the need to provide a list of API call strings. As a consequence, this significantly impedes the malware analysis. However, the malware still has to query this information by accessing, for instance, process internal data structures like the *PEB*. In this case, the malware locates this data structure via the *fs* register on x86, which is a suspicious pattern. The feature $API_{hashing}$ searches for regions that may employ API hashing by matching common access patterns to internal data structures.

② **binary** Often memory regions comprise an executable file, e.g. a *PE* executable on Windows or an *ELF* executable on Linux. Thus, we take features of this executable into account. Please note that the following features yield zero values on memory regions that do not contain executable files.

Programs require API calls to interact with their environment, which is why they import API calls at program start. Malware prefers to resolve APIs at runtime and may not

import APIs at program start. The feature $binary_{imports}$ reflects whether a binary imports APIs or not. Likewise, libraries export functions. In our experience, malware does export significantly fewer functions than a standard system library. For example, the libraries that the malware *Dridex* injects contain less than five exports. On the contrary, the system library *kernel32* exports a couple of hundred APIs. The feature $binary_{exports}$ counts the functions that a library exports.

Binaries comprise headers so as to the OS can load them properly. Hence, if a memory region contains a header then we have found an executable program or library. The feature $binary_{has_header}$ codifies if a memory region contains a header by checking for common magic numbers like the PE magic number *0x4D5A*. Malware may delete its file header to complicate its analysis, e.g. to find it in memory in the first place. It overwrites the first hundred bytes with zero. However, this is detectable if a memory region starts with zero bytes that are followed by code. *Quincy*'s feature $binary_{wiped_header}$ checks this.

Binaries may be programs or libraries. The feature $binary_{is_pe_or_dll}$ reflects whether it is an executable or a library like a *PE* or a *DLL* on Windows. The OS keeps track of all modules in a process space, i.e. we define modules as a program or a library. Malware may inject code that is not known as a module. The feature $binary_{is_module}$ encodes whether a memory region comprises a known module or not. Libraries can be directly loaded at program start or during runtime. The operating system keeps track of the libraries that have been loaded before program start and at runtime. Malware typically injects libraries at runtime via APIs like *LoadLibrary*. For example, the malware *Conficker* injects its library in this way [46]. The feature $binary_{is_dynamic_library}$ encodes the fact that a library was loaded at runtime and may aid in detecting *Conficker*-like injections.

③ **code** The main goal of an HBCIA is to inject *code* into a victim process. Thus, the code properties of memory regions are worth scrutinizing.

To begin, we roughly classify regions into data and code regions. We assume that every meaningful piece of assembly code is modularly structured and therefore split into several low-level assembly functions. Such functions have a prologue and an epilogue, where, for example, local stack frames are set up or cleaned up [272]. The feature $code_{functions}$ scans memory regions for common assembly code patterns of function prologues to detect regions with code.

Once malicious code has been injected into a victim process, it first conducts its initialization. At first, it does not exactly know at which memory address it has been loaded. This knowledge is crucial to properly address data and code. There are several ways that malware can determine its address in memory. For instance, the assembly sequence *call +5; pop eax* just calls the next instruction, which is *pop eax*. However, on x86 the

call instruction pushes the *eip* register to the top of the stack. The instruction *pop eax* pops the stack's first element to *eax*. Thus, this code sequence yields the current address in the register *eax*. Note that a x86 program can not directly read or write the program counter *eip* [272]. Therefore, tricks like this are required. The feature $code_{shellcode}$ looks for common shellcode patterns in memory regions that determine the address of the current instruction.

Banking trojans divert the control flow of certain networking APIs in order to manipulate the communication between a user and its bank. For this sake, they employ code hooks [273]. Other malware types also utilize code hooks, e.g. to hide its presence [274]. Unfortunately, detecting code hooks is computationally expensive because it involves code detection, diassembling of code to find branch instructions, and heuristically taking the decision whether or not the branch between two memory regions poses a code hook. For example, the *Volatility* plugin *apihooks* [25] exhibits a runtime of several minutes when analyzing a one gigabyte memory dump on a modern CPU. Therefore, we present a quick hook detection heuristic. There are functions that malware hooks, such as *InternetReadFile* or *PR_Read* [273, 275], to conduct man-in-the-browser attacks. The feature $code_{hooks}$ scans memory regions for strings of hook target functions. While this may lead to false positives, it significantly speeds up the total runtime of the approximation.

④ **cryptography** Encryption and obfuscation of information are important to malware. On the one hand, this allows it to covertly operate since there are no obvious signs like strings. On the other hand, this increases its analysis time during which it can operate unobserved in the wild. For this sake, malware employs encoding, encryption, and hashing schemes. For example, it may encrypt its network communication with *RSA*, encode its configuration with *Base64*, or hash its API names with *MD5*. To further complicate the analysis, malware statically links such algorithms into its binary, instead of calling the OS provided versions. As a consequence, the analyst does not find any traces of relevant API calls. Therefore, this category's features search for constants and strings of encryption ($crypto_{cipher}$, *0x9e3779b9* from *TEA*), encoding ($crypto_{encoding}$, the *Base64* alphabet), and hashing ($crypto_{hashing}$, *0x67452301* from *MD5*) schemes.

⑤ **countermeasure detection** As long as malware has not been analyzed, it can operate undisturbed in the wild. The malware author wants to postpone the analysis as long as possible for this reason. Therefore, malware is equipped with methods to passively (e.g. code obfuscation [276]) or actively (e.g. DoS attacks against hypervisors) hinder its analysis. The features of this category detect such behavior.

Today, sandboxes carry out an initial analysis of malware before human analysts to effectively cope with the flood of malicious samples [52]. Only special cases are forwarded

to human analysts. For instance, unseen samples that do not fit in any previously known cluster. Therefore, malware may detect that it runs in a sandbox and may show a different behavior [253]. The feature $counter_{sandbox}$ scans memory regions for strings and code patterns that aim at sandbox detection.

To dig deep into malicious code, malware analysts utilize debuggers. There is a plethora of ways how a program can detect that it is being debugged [49]. The feature $counter_{debugger}$ searches for strings and code patterns that allow debugger detection. Typically malware analysts employ disposable VMs as analysis environment so as not to infect their real system. VMs are not common among typical PC users. Therefore, malware authors do not want their piece of code to be run in a virtualized environment. The feature $counter_{vm}$ detects strings and code patterns that permit VM detection.

⑥ **memory** To make a distinction between malicious and benign memory regions, the features of such regions themselves become relevant. This category comprises more features than any other category (13 features), which we grouped in three subcategories: statistical features, memory region features, and strings.

statistical features Memory regions allocated for the first time may just contain a lot of zeros. There are also functions like *ZeroMemory* that initialize a memory region before initial use. Sometimes huge memory regions are requested just to store small objects, leaving the rest untouched resulting in zero, e.g. a one megabyte heap of a new thread with only two small objects. These areas do not contain much data and hence they are not interesting. The feature $memory_{is_sparse}$ targets close to empty memory regions. It computes the ratio of zero bytes to non-zero bytes.

In contrast to $memory_{is_sparse}$, $memory_{high_entropy_areas}$ detects compressed or encrypted areas. Malware often, for example, contains encrypted configuration files or is compressed. Therefore, this feature utilizes entropy analysis to detect encrypted/compressed areas [63]. It splits a memory region into several smaller areas and computes for each area the entropy. If an area exceeds a threshold, it is assumed to be a high entropy area that most likely holds encrypted/compressed data. We chose the area size of four kilobytes since this is the typical page size on the x86 architecture [272] and the threshold of 6.5 as suggested by Lyda et al. [63]. The final result of this feature is the ratio of high entropy areas to all areas of the memory region.

memory region features Another set of interesting features are the actual properties of a memory region. These are, for example, OS assigned flags.

The protection of a memory region tells a lot about its purpose. It can be a combination of *read* (R), *write* (W), and *execute* (X). For instance, the *.text* section of a binary exhibits *RX* protection, i.e. it is just readable and executable, but not writable, which is sufficient

for code to be executed. Many malware families like *Andromeda*, *Goznym*, and *Urlzone* allocate *RWX* memory regions in their victim processes. There are many reasons for this behavior. For example, to inject code it must be writable to be copied over and readable, as well as executable, to properly execute. If the malware directly allocates memory as *RWX* then it does not need any further changes to memory protection. This behavior may suggest that malware authors are rather lazy. But also the architecture of the malware may require this. One example is shellcode that mixes code and data like the banking Trojan *Goznym*. $memory_{protection}$ encodes the memory protection of a region. This is also the main feature of *Malfind* [66].

Memory mapped files allow programmers to handle files on disk as if they were mapped to memory [266]. Hence, the programmer uses common memory operations and does not have to directly access the file via *fread* and related functions. The operating system abstracts the file access for the programmer. On Windows, files can be mapped to a process with *MapViewOfFile*. Memory mapped files are utilized to copy code from the attacker to the victim process. Since memory mapped files can be mapped to other process spaces, malware families like *Stuxnet* employ this technique [12]. The feature $memory_{mapped}$ encodes whether or not a memory region represents a memory mapped file. Memory regions that should not be shared with other processes can be marked as private [266]. Such regions are unlikely to be utilized in a code-injection technique employing memory-mapped files. The feature $memory_{private}$ describes if a memory region is private.

$memory_{is_heap}$ encodes if a memory region is a heap. This feature may help detecting malware in combination with $code_{functions}$, since there are unpackers that decompress code to the heap [252]. On Microsoft Windows, the memory manager tags memory regions according to their internal type [277]. The feature $memory_{tag}$ encodes the different types.

Code-injecting malware requires an execution context for its injection payload. This can be either a thread that is hijacked or a newly created thread (see Section 4.3.3). If a thread originates in a memory region then this is a strong indicator that this memory region contains code. The feature $memory_{thread}$ marks memory regions where a thread originated. This feature is especially interesting in combination with other features, e.g. threads that originate outside of non-module areas like the heap ($memory_{is_heap}$).

strings Text strings help reverse engineers to quickly understand the behavior of a binary. There are typically hundreds of strings in a binary, including many false positives, i.e. byte sequences terminating with a zero byte, which resemble C-style strings. Since strings help malware analysts to quickly understand program behavior, they are often

obfuscated, e.g. via encryption [278]. However, strings are usually decrypted during the unpacking phase and lie unprotected in memory.

It is important for code-injecting malware to identify its victim processes. Even though there are many ways to identify processes on Windows (see Section 6.1.4), our experience has shown that the most common way is comparing the running process names to an internal list of victims. Therefore, the feature $memory_{victim_strings}$ searches for common names of victim processes like *explorer.exe* and *firefox.exe* in memory regions (see Section 5.2).

Data exfiltration is a common theme in malware. Be it a banking Trojan that exfiltrates credit card credentials or be it an espionage malware that exfiltrates construction plans. Network communication is a popular way to exfiltrate data. Hence, strings related to common network protocols like *http* suggest that a memory region may communicate via the network. The feature $memory_{network_strings}$ scans memory regions for such strings like *GET* and *ftp*.

Once a malware runs on a system, it needs to ensure persistence to continuously operate. The malware ensures that it is automatically started at system start, e.g. as a service or via an autostart key in the Windows registry [50]. The feature $memory_{persistence}$ scans memory regions for strings related to persistence, e.g.
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run.

(Benign) programs include text descriptions, e.g. a help about the command line interface or explanations about some feature the user should interact with.
$memory_{english_strings}$ finds common English strings within a memory region. This feature should help to detect benign regions, e.g. program descriptions in a benign executable. Due to this, we compiled a list of the 1000 most common English search terms on *Google* that have three or more characters . The list is based on the list of the 10,000 most popular search terms on *Google* [279]. We opted for words that have three or more characters in order to evade false positives from one and two character sequences. For instance, the word "a" matches 0x4100 that would also match the suffix of the word "Tequila". The longer the word, the lower the probability of false positives.

⑦ **trojan** We described in Chapter 4.2.1 that one advantage of HBCIAs is the ability to intercept critical information from within the victim process. This is especially relevant to (banking) Trojans such as *Nymaim*, *Urlzone*, or *Zeus*. The feature $trojan_{banking}$ searches memory regions for a list of financial terms and names of banks. Another class of data that is frequently stolen are cookies. Hence, the feature $trojan_{cookies}$ scans memory regions for strings related to cookies. To access user accounts, cyber criminals need to know the corresponding credentials. Therefore, many Trojans implement key logging

techniques to steal credentials. The feature $trojan_{credentials}$ looks for, e.g. names of social media websites like *Facebook*, in memory regions.

7.2.3 Feature Selection

We have presented 36 different features from seven categories in the previous section. Many features are binary, e.g. $memory_{is_heap}$ or $binary_{is_dynamic_library}$. But there are also continuous features, such as $memory_{is_sparse}$ and $memory_{high_entropy_areas}$. Since some machine learning algorithms like *SVM* require standardized features [82], we apply the standard score to the features. It is given by $f' = \frac{f-\mu}{\sigma}$, where μ is the mean and σ is the standard deviation of feature f [86]. The mean of the standard score is always zero and its deviation is one.

We could have directly applied machine learning algorithms to these 36 features, which span a 36-dimensional feature space. However, there are several reasons why it is advisable to reduce the feature set size and utilize less features. If we reduced the feature set size and shrinked it to a set with only relevant features then this would be beneficial. For instance, this would yield faster model training due to less dimensions and model simplification due to removal of irrelevant or redundant features. Therefore, we chose to employ feature selection.

We employed *Recursive Feature Elimination* (*RFE*) [280]. It recursively trains models with smaller and smaller feature sets, where it cuts back on the least important feature in each recursion step until it encounters the optimal set. *RFE* requires an external estimator that is capable of ranking features due to their importance. Genuer et al. [281] recommended *Random Forests* as external estimator. We followed their recommendation.

7.2.4 Training and Classification

This section describes the training and classification of our system, which is based on machine learning. First, we discuss requirements for a machine learning algorithm. Second, we argue the choice of the algorithms used to evaluate our system.

Requirements for the Machine Learning Algorithms

As we have already stated, the choice of a machine learning algorithm depends on many aspects. Often data scientists have to go through several trial and error rounds to find

a good candidate for their problem. Rohrer [282] discussed several considerations that we address in the following as requirements **R1** to **R4**.

R1 Training and Testing Complexity As with every other algorithm, the computational complexity of a machine learning algorithm is a non-negligible factor when choosing it, especially when dealing with huge data sets. When it comes to machine learning algorithms, two computational complexities are of interest. First, the training complexity, and second, the testing complexity of a new sample.

R2 Non-Linearity Linear algorithms assume that decision boundaries can be represented by a linear function (e.g. $3.5x + 100$). However, the decision boundary of many problems can not be adequately represented by a linear function. Linear algorithms can be trained faster than non-linear algorithms. Our problem is a non-linear one (see Section 7.3.1) and hence we require an algorithm that can handle such problems.

R3 Number of Parameters The parameters of a machine learning algorithm have great impact on its prediction efficiency. The more parameters an algorithm offers, the better the algorithm can be tweaked to one's needs. However, tweaking the parameters too much yields a higher risk of overfitting. Therefore, we prefer algorithms with fewer parameters.

R4 Number of Features Some machine learning algorithms do not scale well with an increasing number of features. This phenomenon is known as *The Curse of Dimensionality* [85], since each new feature adds a new dimension to the feature space, in which the algorithm works. This may dramatically slow down the computational performance of the algorithm.

Discussion of the Machine Learning Algorithm Choices

There is a plethora of machine learning algorithms from which we may choose. Not every algorithm works well for every problem. We drew candidates from a widely cited machine learning survey by Caruana et al. [283] and also added candidates that the widely utilized machine learning library *scikit-learn* implements [267]. Table 7.3 matches our requirements to several supervised machine learning algorithm classes.

To begin, we discuss why we discarded some of the classes. The *Linear Regression Analysis* class does not work well on non-linear data. Since our data set was not linear-separable (see Section 7.3.1), we discarded this class. The *Naive Bayes* class assumes features to be independent [284], which was not the case with our feature set, e.g. the

Machine Learning Algorithm Class	**Representatives**	**R1**	**R2**	**R3**	**R4**
Decision Trees	CART	●	●	◐	●
Randomized Tree Forests	*RandomForest, Extremely Randomized Trees*	◐	●	◐	●
Tree-based Boosting	*AdaBoost, GradientBoostingMachine*	◐	●	◐	●
Support Vector Machines	*RBF SVM*	◐	●	●	◐
Nearest Neighbors	*K-Nearest Neighbors*	◐	●	●	◐
Neural Networks	*Multi-layer Perceptron*	○	●	○	○
Linear Regression Analysis	*Logistic Regression*	●	○	◐	●
Naive Bayes	*Gaussian Naive Bayes*	●	◐	●	○

TABLE 7.3: Matching of supervised machine learning algorithm classes to our requirements. We listed at least one representative per class. Three possible results: does not match (○), partially matches (◐), and matches (●).

features $api_{general_api_strings}$ and api_{hbcias}. Optimal *Neural Networks* are difficult to build. Furthermore, they have problems with a larger number of features (Curse of Dimensionality) [85] and their output is difficult to comprehend. We did not consider these algorithm classes in the following. Please note that we discuss *Neural Networks* as a possible research branch in future work (see Section 8.2.3).

We selected the following machine learning classes and some of their representatives for our evaluation of *Quincy*. *Nearest Neighbors* is a simple but powerful class that exploits neighborhood relationships to classify samples. *K-Nearest Neighbors* [84], as one of its implementations, works with non-linear data and it has only two parameters (the number k of neighbors to consider and its distance metric). Additionally, it has a fast training time of $\mathcal{O}(1)$ since it just stores all samples. The tree-based algorithm classes *CART Decision Trees*, *Randomized Tree Forests*, and *Tree-based Boosting* closely match our requirements. Therefore, we evaluated *Quincy* with several algorithms from these three classes. We opted to evaluate *CART-Decision Trees* [87], *Random Forests* [89], *Extremely Randomized Trees* [90], *AdaBoost* [285], and *GradientBoosting* [286]. *Support Vector Machines* are capable of dealing with non-linear data when employed with, for instance, a RBF kernel [82]. They only have two hyperparameters (C and γ). However, they suffer from the Curse of Dimensionality. Nevertheless, we included one of their implementations with RBF kernel since we employed a feature selection that reduced the size of the feature set.

7.2.5 Implementation

We implemented a prototype of *Quincy* and published it on the code portal *github* [29]. We chose Python as implementation language since it enabled us to quickly prototype. The prototype employs *Volatility* [25] to close the semantic gap and to extract the features. This permits *Quincy* to analyze memory dumps of all Windows NT versions since

Windows XP. Its machine learning heuristic is based on the library *scikit-learn* [267], which implements common machine learning algorithms.

7.2.6 Discussion of Evasion

Every detection system is subject to evasion. Machine learning-based systems are not an exception [287]. The goal is to raise the bar reasonably high to make evasion harder or at best practically infeasible.

Current systems such as *Malfind* and *Hollowfind* can be evaded with ease. *Malfind* only relies on memory region protection and obvious unlinking of modules from a process internal data structure. Hence, it can be evaded by setting proper memory region protection and not unlinking modules, e.g. as in the case of the malware *Ponmocup*. *Hollowfind* focuses on *Process Hollowing*, a subtype of HBCIAs. Therefore, not using *Process Hollowing* evades it. The evaluation result in *Membrane*'s paper [28] shows that *Membrane* is very prone to noise. A possible way to hide injected code would be to increase the noise by creating many new memory regions with random content.

Quincy employs more features than previous systems and it also has a broader focus than *Hollowfind*. Its models are based on up to 36 features. To evade our system, an attacker has to adjust its malware to bypass all features. Bypassing one single feature may be easy, e.g. bloating a memory area with many zeros to attack the feature $memory_{is_sparse}$. However, bypassing feature combinations is harder, especially opposing features such as $memory_{is_sparse}$ and $code_{functions}$. Therefore, our system raises the bar for an attacker significantly when compared to previous systems.

7.3 Evaluation

We evaluate *Quincy* in this section. At first, we discuss the data set that we utilize in this evaluation. Next, we describe how we conducted the evaluation. Finally, we discuss the evaluation results.

7.3.1 Data Set

An evaluation is only as relevant as its corresponding data set. Unfortunately, there are only a handful of scientific open source malware data sets, which were not useful to us. In this section, we discuss how we compiled the evaluation data set and what kind of data it contained. First, we present the requirements for a data set to evaluate a static

HBCIA detection system on memory dumps. Subsequently, we present the malware and goodware samples that we utilized to generate memory dumps. Then, we describe how we ensured a proper ground truth. Lastly, we performed a first data analysis to have a first impression of what kind of data we would face.

Requirements

Unfortunately, related work did not define any requirements regarding an evaluation data set. Furthermore, the Windows corpus of Section 6 did not serve in this evaluation because we intentionally left out HBCIA-employing malware that utilized *Process Hollowing*. This was owing to the fact that *Bee Master* can not detect this technique. But *Quincy* does not suffer from this limitation. Furthermore, we wanted to add even more HBCIA representatives and goodware samples as well as evaluate *Quincy* on the latest Windows version 10. Therefore, we could not utilize this data set here. Consequently, we needed to build a new data set for this evaluation. We define requirements for a data set in the following as in [17].

First of it all, the set has to comprise a considerable amount of HBCIA-employing malware families to ensure an evaluation of distinct code injection techniques (**R1**). We wanted to ensure that the evaluation results applied to current malware. This means that we considered recent malware families that were still found in the wild (**R2**). The main target for malware is still Windows (e.g. [4]). Hence, we wanted malware that targeted this operating system (**R3**). Lastly, an HBCIA detection system for memory dumps encounters malware but also by orders of magnitude more goodware. Therefore, this amount of goodware should be reflected in the data set. As a result, we needed it to contain goodware (**R4**).

To summarize our requirements:

R1 a considerable amount of HBCIA-employing malware families

R2 recent malware families

R3 only Windows malware

R4 goodware programs to estimate false positives

Table 7.4 compares our requirements to six open source malware data sets. Whereas *Android Malware Genome Project*, *Drebin*, and *Android Malware Dataset* only offer Android malware and therefore violate **R3**, *cwsandbox* and *malicia* comprise obsolete families and hence violate **R2**. The *Malware Classification Challenge* set is comprised of a great

Data Set	Year	Publication	R1	R2	R3	R4
cwsandbox	2007	[52]	✗	✗	✓	✗
Android Malware Genome Project	2012	[238]	✗	✗	✗	✗
Malicia	2013	[239]	✗	✗	✓	✗
Drebin	2014	[240]	✗	✗	✗	✗
Malware Classification Challenge	2015	[241]	✗	✓	✓	✗
Android Malware Dataset	2017	[242]	✗	✓	✗	✗

TABLE 7.4: Comparing open source malware data sets with our four requirements to an evaluation data set.

amount of malware samples. Nonetheless, these samples just split up into eight families. Some of them do not employ HBCIAs. Unfortunately, none of the open source malware data sets matched our four requirements. Therefore, we decided to compile our own data set that satisfied our requirements. We published this set on the code portal *github* [30] to foster open source research.

Goodware and Malware Samples

This sample set differed from the set in Section 6. First, it contained more families including families that utilize *Process Hollowing*. Second, we added many new families from 2016 and 2017. Third, we added more goodware samples. Our data set comprised 1794 goodware and 102 malware samples. The 102 malware samples belonged to 102 distinct malware families. We showed in Section 5.3 that it is sufficient to take a representative of an HBCIA-employing malware family since it is a family-inherent feature.

The goodware samples of our data set contained Windows system programs and other widespread freeware programs. For this purpose, we gathered system programs from Windows XP, 7, and 10. Further, we downloaded widespread freeware programs from an archive of portable freeware applications [288]. These programs included, for example, browsers (Firefox), mail clients (Thunderbird), chat clients (Pidgin), and cryptographic software (Truecrypt). This added up to 1794 goodware binaries. Please note that the total number of binaries per Windows version varied due to the fact that Windows system programs are only compatible with one Windows version. We have provided a hash list of all binaries on *github* [17] due to space constraints of this thesis.

The malware samples of our data set comprised 102 representatives of code-injecting malware families. We showed in Chapter 5.3 that HBCIAs are an inherent malware family feature. It suffices to pick one representative per family. The main idea behind this was that we shunned overfitting since we did not train the algorithm to detect a

certain family but rather the signs of code injections presented by many distinct families. The set of HBCIA-employing malware representatives was created in the course of this thesis. We collected the samples from various sources: antivirus companies, IT security blogs, and an internal malware archive. In all cases, we verified the code-injecting behavior of the malware family by manual analysis. Please note that some samples did not execute on every Windows version. The reasons for this are manifold, e.g. the malware was especially developed for a Windows version or security mechanisms on newer versions prevent proper execution. As a consequence, the number of malware families per Windows version varied in the evaluation. We have provided all malicious binaries on *github* [17].

Memory Dump Generation

Quincy detects HBCIAs in memory dumps. More precisely, it detects malicious memory regions in memory dumps. A memory dump contains thousands of memory regions, distributed over several virtual process spaces. To obtain memory regions in order to train and classify, we had to generate memory dumps for each sample. Since *Quincy* is capable of analyzing various Windows versions, we generated a memory dump of each sample for Windows XP SP3, Windows 7 SP1, and Windows 10.

We automated the generation of memory dumps. For this sake, we implemented a program based on the virtualization framework *VirtualBox* [72]. The generation process worked as follows: First, the program built an ISO disk image that comprised the sample. Next, it started *VirtualBox* in a predefined state. This was followed by mounting the ISO disk via the virtual DVD drive. We placed a shell script in the guest system that waited for new virtual DVDs and executed the sample with administrator privileges. Subsequently, the program waited for two minutes to allow the sample to initialize and conduct its code injection. Note that two minutes is a common timeout in sandboxing systems. This is based on domain knowledge. To the best of our knowledge, there has been no evaluation of this parameter. Finally, the program asked *VirtualBox* to create a memory dump and to save it to a file.

A problem may be environment sensitive malware [253], which tries to detect the analysis environment and refuses to work properly. We hardened the VMs to overcome the problem of environment sensitive malware. This included, for instance, the removal of hypervisor related strings from the Windows registry or the deinstallation of VM tools. Hardening involved the tool *Pafish* [289], which implements several tests to detect hypervisors. Furthermore, we did not grant the VMs access to the Internet since no contact

OS	Binaries		Memory Regions	
	Goodware	Malware	Benign (All/Unique)	Malicious
Windows XP	1205	71	2,729,563/15,919	398
Windows 7	1264	72	5,306,368/32,005	319
Windows 10	977	73	7,266,226/129,786	710

TABLE 7.5: Summary of the three data sets of Windows XP, 7, and 10. Benign memory regions may be duplicates, e.g. system libraries. The amount of memory regions constantly increases from version to version.

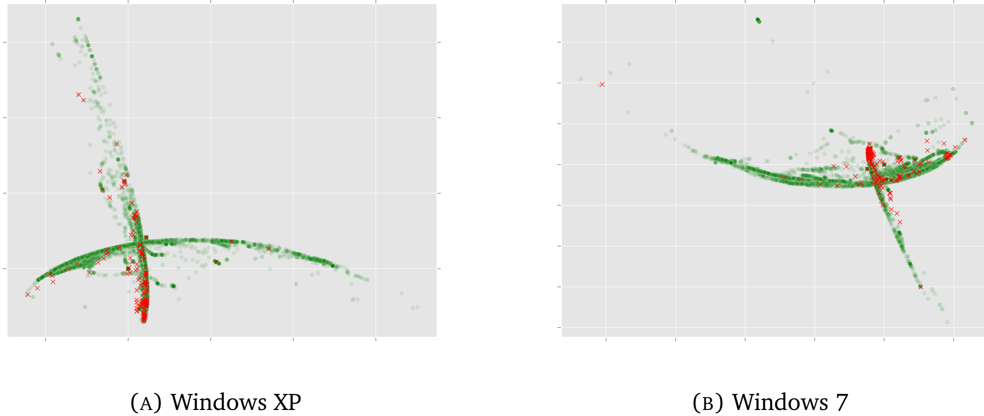
to any command and control infrastructure was required in order to conduct a code injection.

Ground Truth

A sound ground truth is crucial for the later evaluation. It is required to reliably compare *Quincy* to its two contenders *Malfind* [25] and *Hollowfind* [27]. We constructed this ground truth by the following rules. First, we assumed all memory regions of goodware dumps as benign. Such an assumption could not be stated for malware dumps. Their memory regions are either malicious or benign. As a solution to this problem, we utilized *YARA* signatures [290] to detect the malicious memory regions of the malware dumps. To create *YARA* signatures, we manually reverse engineered the malicious binaries and extracted a signature based on the malware's content in memory. Unfortunately, there was only previous work on automatic signature generation of binaries [291]. However, the malicious binary and the malicious memory regions may significantly differ. For instance, due to executable packing [55]. Hence, this previous work was not applicable to our problem and we had to tediously reverse engineer each malicious binary to ensure a sound ground truth. All memory regions of a malicious dump that did not match a signature were assumed to be benign. To the best of our knowledge, we were the first to employ such an approach to label benign and malicious artifacts for a scientific evaluation.

Data Analysis

We conducted an initial data analysis to provide a first impression of the data set. This should help in understanding the choice of machine learning algorithms and the later performance of the detection systems on the individual Windows data sets. Table 7.5 summarizes the data set of Windows XP, 7, and 10. First of it all, the distribution of benign and malicious binaries was heavily skewed. The reason is that it is harder to properly gather representatives of HBCIA-employing malware families than to gather



(A) Windows XP

(B) Windows 7

FIGURE 7.2: Isomaps of Windows XP (left) and Windows 7 (right). Green dots denote benign samples, red crosses denote malicious samples. The samples scatter along two main axes.

benign executables. When we compared resulting memory regions, we noted that the distribution was even more skewed. There were millions of benign memory regions compared to a couple of hundreds malicious ones. However, the benign memory regions contained duplicate data, e.g. system libraries that were present in all dumps. We opted to remove these duplicates to decrease the amount of data that the machine learning algorithms had to handle. After removing duplicates, the relation between benign and malicious memory regions was much closer. The benign memory regions were a multiple of the malicious ones in the range from 40 (Windows XP) to 182 (Windows 10). Please note that a duplicate pair may not have the same hash value, it just exhibits the same numerical properties in the feature space.

Visualization of high dimensional data is difficult but it is possible through dimensionality reduction. There are several algorithms that transform high dimensional data to low dimensional data [82]. They attempt to preserve the relationship between the individual samples. Examples are *principal component analysis* (PCA) [82], *locally linear embedding* (LLE) [292], and *isometric feature mapping* (Isomap) [293]. We opted to utilize *Isomap* because it is capable of preserving neighborhood relations between samples [293]. *Isomap* consists of three steps: First, it creates a weighted graph that represents the neighborhood relations between the samples in the original space. *KNN* is utilized to determine these relations. Second, it creates a matrix of distances between each sample. Third, classical *multidimensional scaling* (MDS) like PCA is applied to this matrix to reduce the dimensionality.

Figure 7.2 shows the isomaps of the Windows XP and Windows 7 data sets. Malicious samples are illustrated by red crosses and benign samples by green circles. The majority of samples scatters on two main axes, which cross each other. Most malicious samples

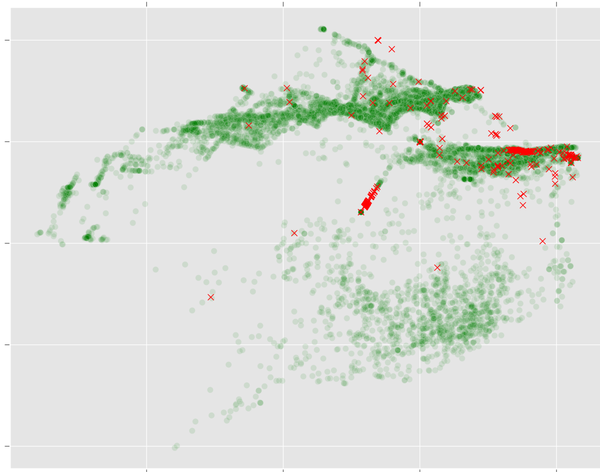


FIGURE 7.3: Isomap of Windows 10. Green dots denote benign samples, red crosses denote malicious samples. The samples are broader scattered than on Windows XP and Windows 7.

scatter at the end of one axis. However, there are malicious samples scattered along both axes, leading us to conclude that there is no simple boundary between the malicious and benign samples. Hence, there is no linear separation and linear machine learning algorithms may not perform well on our problem. Figure 7.3 shows the isomap of Windows 10. The samples are more broadly scattered than in the case of Windows XP and 7. This holds true for malicious and benign samples. Hence, we assumed that detection of malicious samples on Windows 10 was harder than on Windows XP and 7. Additionally, we had to consider that the ratio between malicious and benign samples was significantly lower than on Windows XP and Windows 7.

7.3.2 Methodology

Finding the best model to solve a problem at hand is not an easy task. A solution is to consider several models and to select the most appropriate one for the problem. One major obstacle to overcome is the optimization of the machine learning algorithm's parameters to achieve (near) optimal predictive performance. Yet the scientific machine learning community acknowledges that a trial and error process is the most feasible way [268].

Figure 7.4 shows the four steps of the evaluation. The evaluation was carried out for each Windows version separately. We enclosed the evaluation in a 10-fold cross validation loop to cope with variance. Cross Validation splits the sample data into several

complementary data sets. Multiple validation rounds on these data sets reduce the variance and increase the data analyst's confidence in the model. A popular Cross Validation approach is *k-Fold Cross Validation*. In k-Fold Cross Validation, the data set is split into k equally sized data sets. Then in k rounds, (k-1) data sets are used for training and one data set is used for validation. During the k rounds, each of the k data sets is used exactly once for validation. Hence, the algorithm validates the model on all data. Even though the parameter k can be freely chosen, a common choice for k is 10 [294].

① First off, we randomly created two data sets d_{train} and $d_{validate}$ from the benign and malicious data sets. d_{train} was utilized to train and optimize our machine learning models. It contained 60% of the malicious and 10% of the benign samples. $d_{validate}$ was our validation data set. We evaluated the final performance of the optimized models on this data set. It contained 40% of the malicious and 90% of the benign samples. As we showed in Section 7.3.1, the distribution between malicious and benign memory regions was heavily skewed. Therefore, we opted to treat the two data sets separately. Since many machine learning algorithms tend to misclassify the minority class [295], we mitigated this effect by putting fewer benign and more malicious samples in d_{train}. Having more benign samples in the validation data set also simulated the noise that detection algorithms encounter in a real world setting. Actually, the seminal paper of *Membrane* [28] showed how increased noise can drastically decrease the performance of an HBCIA detection system.

Furthermore, we ensured that all memory regions of a malware family were either exclusively in d_{train} or exclusively in $d_{validate}$. This ensured that we did not validate our models on known malware families. We wanted to detect the code injections rather than a certain malware family.

② After splitting the data sets, we selected the optimal set of features on d_{train}. We employed a *Recursive Feature Selection* (RFE) [280]. We described the exact process in Section 7.2.3. The resulting optimal feature set was utilized in the cross-validation iteration during training and validation.

③ Machine learning algorithms may require several input parameters. They are called *hyperparameters*. An example of a hyperparameter is the maximal number of classifiers in a learning ensemble. Since these hyperparameters have a great impact on the final performance of the induced model, their optimization is a crucial step towards the optimal model.

The first step of such an optimization is to choose the hyperparameters of relevance for the performance and a range of possible values. The space that the hyperparameters of a machine learning algorithm span is called the *hyperparameter space*. This space

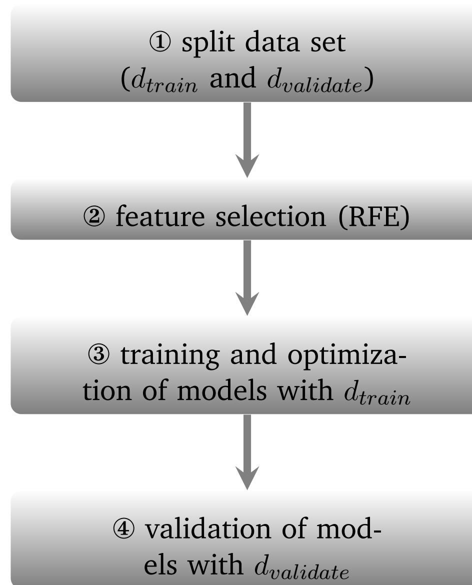


FIGURE 7.4: Flow chart of one fold of the 10-fold cross validation loop.

can be infinite. Hence, we had to define a finite subspace to search for our optimal hyperparameters. Please note that each hyperparameter the user wishes to optimize adds a dimension to the hyperparameter space and consequently increases optimization time.

Once the hyperparameters were chosen and their corresponding hyperparameter space was defined, they could be optimized. The basic algorithm is called *Grid Search*. This algorithm exhaustively searches through a manual defined subspace of the hyperparameter space [296]. It determines the performance of a certain hyperparameter combination with cross validation given a metric. This algorithm finds the optimum with respect to the space it searches through. However, its main drawback is its computationally expensive runtime since it suffers from the Curse of Dimensionality [296]. Another approach is *Randomized Grid Search* [296]. Instead of exhaustively searching through the hyperparameter space as *Grid Search* does, *Randomized Grid Search* randomly chooses grid points from this space and evaluates their performance. Bergstra et al. [296] showed that if the same time is granted to *Grid Search* and *Randomized Grid Search* then *Randomized Grid Search* is likely to perform as well as *Grid Search* – if not better – since it can search through a larger hyperparameter space in the same time frame.

Bergstra et al. [296] suggested sampling 64 grid points [296]. We evaluated each grid point with a 10-fold cross validation to cope with variance. The final hyperparameter combination was the combination that evaluated using the highest ROC AUC score [82]. The x-axis plots the false positive rate given by $\frac{false\ positives}{false\ positives\ +\ true\ negatives}$ and the y-axis the true positive rate given by $\frac{true\ positives}{true\ positives\ +\ false\ negatives}$. Both axes are typically scaled to the range of [0,1]. The diagonal from [0,0] to [1,1] represents random guessing. The

Algorithm	Trees	Learning Rate	Max. Features	Tree Depth
AdaBoost [285]	[10,100]	[0.1,1.0]	-	-
CART [87]	1	-	$\sqrt{\lvert f \rvert}$, $\lvert f \rvert$	[3,12] + ∞
Extremely Randomized Trees [90]	[10,100]	-	$\sqrt{\lvert f \rvert}$, $\lvert f \rvert$	-
GradientBoosting [286]	[10,100]	[0.1,1.0]	-	[4,8]
Random Forest [89]	[10,100]	-	$\sqrt{\lvert f \rvert}$, $\lvert f \rvert$	-
Algorithm	**k**	**Metric**	**C**	γ
KNN [297]	$2^x, x \in \{1, .., 6\}$	[uniform, distance]	-	-
SVM [83]	-	-	$\{2^{-5}, 2^{-3}, 2^{15}\}$ [298]	$\{2^{-15}, 2^{-13}, .., 2^3\}$[298]

TABLE 7.6: Summary of the algorithms' hyperparameters that we chose to optimize. The total number of features is denoted by f. ∞ denotes that there was no depth limit, the algorithm determined the optimal depth.

perfect model tangents the top left corner ([0,1]). In general, the more a curve tends towards the top left corner, the better the performance of the model. The *area under the curve* (AUC) is called ROC AUC score.

Table 7.6 lists the hyperparameters that we evaluated. Since hyperparameter optimization is computationally expensive, we only evaluated those parameters we thought would have the highest impact on the overall result. The remaining parameters were set to their default values with respect to the employed machine learning library *scikit-learn* [267].

④ Finally, we validated the optimized models on $d_{validate}$ and compared them with each other as well as to *Malfind* and *Hollowfind*.

7.3.3 Results

This section discusses the results of our evaluation of *Quincy*, *Malfind*, and *Hollowfind*. First, we discuss the results of the feature selection and point out which features were more relevant to the detection of malware in memory. This is followed by the results of the model selection and a discussion of the final algorithm choice. Then, we compare our optimized model to *Malfind* and *Hollowfind*. We continue with a temporal evaluation that examined the question of how the three systems performed on newer malware when trained on older malware. Finally, we conclude this section with a discussion on the final models that were published along with *Quincy*'s implementation [29].

Feature Selection

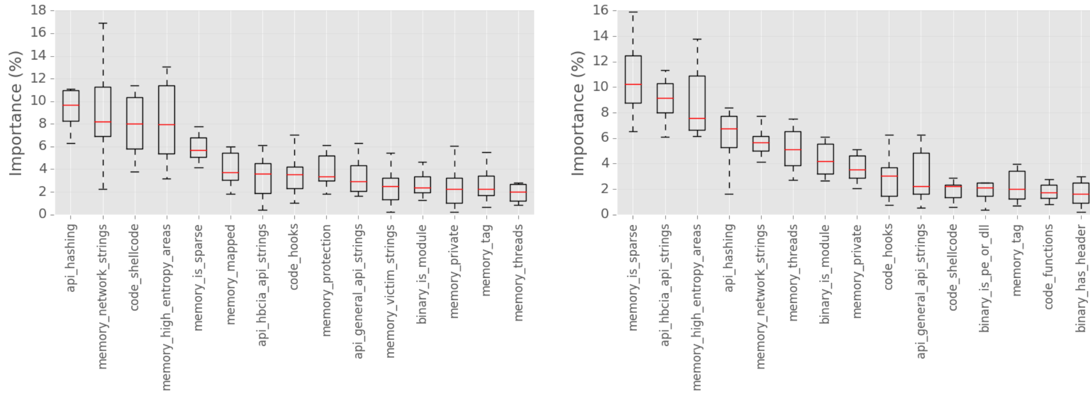
The feature selection had two objectives. First, duplicate or irrelevant features were removed. Second, the reduced feature set also decreased the training time. This holds especially true for algorithms like *SVM* that are susceptible to the *Curse of Dimensionality*.

We carried out the selection on d_{train} during each cross validation iteration. Hence, the features varied on the three different Windows versions. Figure 7.5 shows the top 15 features on average on Windows XP, 7, and 10. It shows the averaged feature importance based on the internal ranking of the *Random Forest* classificator, which was employed in the *Recursive Feature Elimination* (see Section 7.3.2). Note that this is not the final selection of features, only the features during model training and testing. The features of the final models are discussed in Section 7.3.3 and listed in Appendix C.

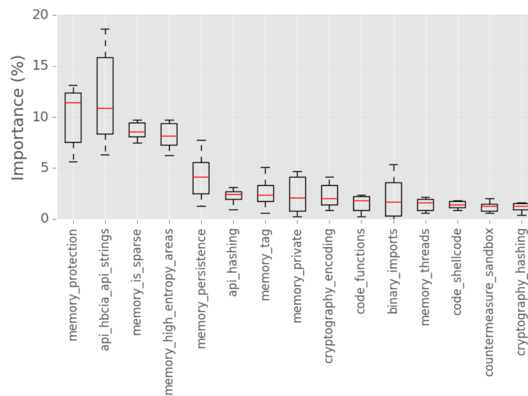
First of it all, the figure shows that there were four outstanding features on all three versions. They reached well over 30% on average of the feature importance. In all cases, they were from the categories *api*, *code*, and *memory*. Interestingly, one of the best performing features was $api_{hbcia_api_strings}$, which finds strings of common code injection techniques like *OpenProcess*, *WriteProcessMemory*, and *CreateRemoteThread*. This shows that most malware resides unprotected in memory. Even though the majority of malware employs code packing [252], once unpacked in memory, for instance, domains and configuration parameters can be found. On Windows 10, there were four especially important features ($memory_{protection}$, $api_{hbcia_api_strings}$, $memory_{is_sparse}$, and $memory_{high_entropy_areas}$) with the other features having less than five percent of importance on average. The features $memory_{protection}$ and $memory_{is_sparse}$ determine together if a memory region contains code. The features $memory_{high_entropy_areas}$ and $api_{hbcia_api_strings}$ target two characteristic traits of malware: compression/encryption and code injections.

Another insight was that the main feature of *Malfind* $memory_{protection}$ was not that relevant to the models on Windows XP and 7. However, its relevance drastically increased on Windows 10. This is in line with the finding of Section 7.3.1, where we stated that the number of *RWX* memory regions is constantly decreasing in newer versions of Windows. This means that allocating a *RWX* memory region is suspicious on Windows 10. Since there were less *RWX* memory regions in the Windows 10 data set, the Windows 10 model focused on this feature and it ranks first. However, this does not mean that it is sufficient to look for memory regions with this feature, since there are families that properly allocate memory regions or inject libraries. This includes, for example, *Feodo*, *Teerac*, and *Ponmocup*.

The category *trojan* was the worst performing category. An explanation is that the data set did not exclusively contain banking Trojans. These features are overspecific and the machine learning algorithms prefer features that are more general to code injections and malware.



(A) Windows XP & 7



(B) Windows 10

FIGURE 7.5: Top 15 features on Windows XP (top left), Windows 7 (top right), and Windows 10 (bottom), ranked by their relative feature importance. We estimated the values during the recursive feature elimination with *Random Forests*.

Model Selection

The model selection process included training and optimization of *Quincy* with several machine learning algorithms on d_{train} on Windows XP, 7, and 10 and the final validation on unseen data of $d_{validation}$. We chose the model that performed the best as final model for *Quincy*. Tables 7.8 and 7.9 list the results of *Quincy* with seven machine learning algorithms. The algorithm ranking is very similar on the three Windows versions, being consistent for the first two algorithms. Table 7.7 summarizes the algorithm ranking.

Simple *Decision Trees* yielded very few false positives but also did not recall as many positive cases as the other algorithms. Hence, they were not a viable option for the final model. The ensemble learning algorithm *AdaBoost* performed well on the training data but only performed better than *Decision Trees* on the validation data. This may be due to overfitting. *SVM* had problems with recalling positive cases. Thus, it yielded many false negatives. Additionally, *SVM* exhibited the longest runtime during training. For

Algorithm	Windows XP	Windows 7	Windows 10
Extremely Randomized Trees	1	1	1
Random Forest	2	2	2
Gradient Boosting	4	4	3
KNN	3	5	5
SVM	6	3	7
AdaBoost	7	6	4
Decision Trees	5	7	6

TABLE 7.7: Summary of the final machine learning algorithm ranking on Windows XP, 7 and 10. The ranking is based on the final evaluation on $d_{validation}$.

these reasons, we discarded *AdaBoost* and *SVM*. *KNN* yielded many false positives on Windows 10, where the malicious and benign cases were more mixed than on Windows XP and Windows 7 (see Section 7.3.1) but it also had a considerable recall of positive cases. Therefore, it ranked third on Windows XP.

Figures 7.6 and 7.8a plot the ROC curves of top five algorithms. The top three algorithms were all tree-based ensemble learning algorithms. They combine several *Decision Trees* to increase performance. They represent two classes: boosting (*Gradient Boosting*) as well as bagging (*Extremely Randomized Trees* and *Random Forest*). The bagging algorithms performed better on our data sets than the boosting algorithm. *Gradient Boosting* reached high accuracy during training but it overfit. Therefore, we discarded it. The bagging-based ensemble algorithms *Extremely Randomized Trees* and *Random Forest* performed very well. *Extremely Randomized Trees* performed better than *Random Forest*, which could be explained by the fact that *Extremely Randomized Trees* introduced even more randomness during training and thus they could keep up to the variance encountered in $d_{validation}$. Therefore, we opted for *Extremely Randomized Trees* as our final algorithm. It ranked first on all three operating systems, being able to cope with variance in data, without overfitting and it offered considerable improvement when compared to the other approaches later.

Comparison to Malfind and Hollowfind

After selecting the optimal models based on *Extremely Randomized Trees*, we compared them to the current state of the art *Malfind* and *Hollowfind*. A description of both systems is provided in Section 7.1. The evaluation was carried out on $d_{validation}$ for Windows XP, 7, and 10.

Table 7.10 summarizes the results on Windows XP, 7, and 10. Figures 7.7 and 7.8 plot the ROC curves of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on

Windows	AdaBoost					Decision Tree					Extremely Randomized Trees					Gradient Boosting				
	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN
XP	84.80%	59.8	56.0	13913.0	25.7	85.84%	62.2	128.5	13840.5	23.3	89.50%	68.0	72.3	13896.7	17.5	86.05%	62.2	107.2	13861.8	23.3
7	76.65%	55.5	132.7	28385.3	49.7	76.04%	55.0	289.7	28228.3	50.2	82.65%	67.7	132.1	28385.9	37.5	78.68%	59.6	214.1	28303.9	45.6
10	74.57%	66.9	103.8	113701.2	72.2	71.80%	58.9	83.6	113721.4	80.2	78.70%	77.6	270.5	113534.5	61.5	74.92%	66.6	446.3	113358.7	72.5

TABLE 7.8: Final data of the evaluation of *Quincy* with *AdaBoost*, *Decision Tree*, *Extremely Randomized Trees*, and *Gradient Boosting* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

Windows	KNN					Random Forest					SVM				
	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN
XP	87.15%	63.8	116.9	13852.1	21.7	87.44%	64.3	54.4	13914.6	21.2	85.29%	61.1	154.4	13814.6	24.4
7	77.67%	57.6	207.8	28310.2	47.6	80.90%	64.0	124.7	28393.3	41.2	78.83%	59.8	180.0	28338.0	45.4
10	74.21%	66.7	326.4	113478.6	72.4	78.53%	77.1	259.1	113545.9	62.0	69.68%	53.2	84.1	113720.9	85.9

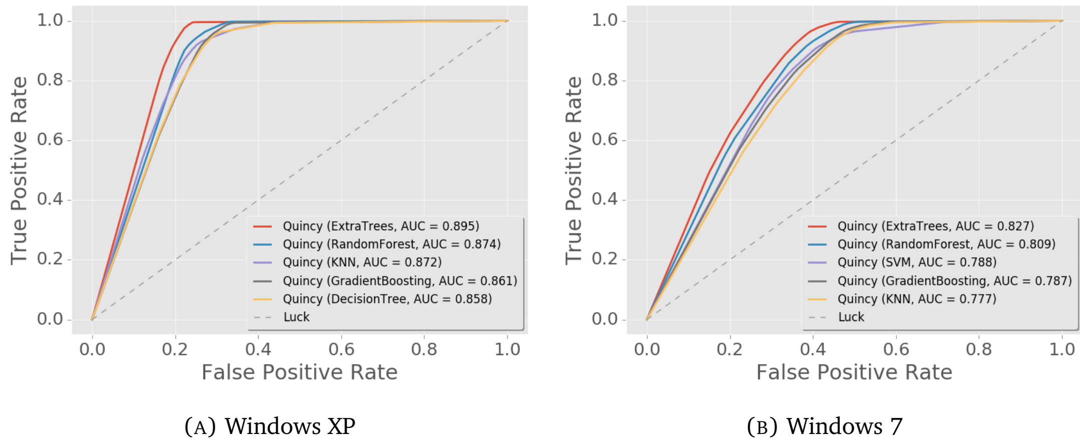
TABLE 7.9: Final data of the evaluation of *Quincy* with *KNN*, *Random Forest*, and *SVM* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

Windows	Quincy					Malfind					Hollowfind				
	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN
XP	89.50%	68.0	72.3	13896.7	17.5	85.80%	62.2	199.4	13769.6	23.3	54.36%	7.7	66.2	13902.8	77.8
7	82.65%	67.7	132.1	28385.9	37.5	73.84%	49.2	141.4	28376.6	56.0	52.93%	8.5	667.0	27851.0	96.7
10	78.70%	77.6	270.5	113534.5	61.5	74.70%	67.1	273.9	113531.1	72.0	52.68%	6.9	91.5	113713.5	132.2

TABLE 7.10: Final data of the evaluation of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

Windows	Quincy					Malfind					Hollowfind				
	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN	AUC	TP	FP	TN	FN
XP	93.32%	68	74	13895	10	89.02%	62	200	13769	16	56.82%	11	64	13905	67
7	86.41%	58	169	28349	21	78.86%	46	142	28376	33	52.62%	6	667	27851	73
10	82.23%	44	263	24	113542	80.76%	42	264	113541	26	53.63%	5	92	113713	63

TABLE 7.11: Final data of the temporal evaluation of *Quincy* with *Extremely Randomized Trees* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).



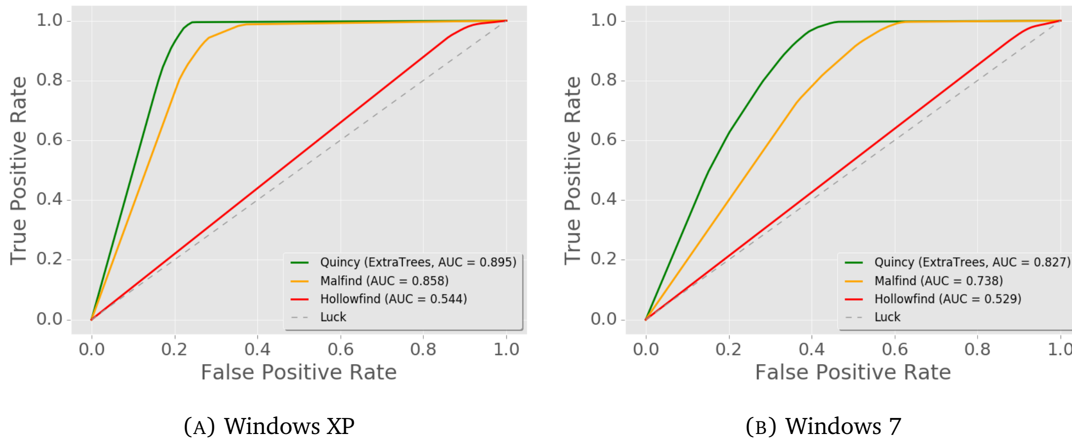
(A) Windows XP

(B) Windows 7

FIGURE 7.6: ROC curves of top five classifiers on Windows XP (left) and Windows 7 (right). *Extremely Randomized Trees* and *Random Forest* rank first on both Windows versions.

the aforementioned operating systems. The listed results are the mean of the 10-fold cross validation described in Section 7.3. Our system dominated the two others on all three Windows versions. *Malfind* dominated *Hollowfind* in all cases. Since *Hollowfind* focuses on Process Hollowing, a subclass of all *HBCIAs*, it only detected a fraction of the samples. It did not exceed more than 54% accuracy. Hence, *Hollowfind* is no longer discussed.

Quincy and *Malfind* achieved the best results on Windows XP, which is the operating system with the least noise (see Section 7.3.1). Notably, our system decreased false positives by 63.74%, when compared to *Malfind*. This saves a lot of analyst time since they do not have to follow dead ends. Many families that run on Windows XP allocate *RWX* regions, which allowed *Malfind* to keep up. However, once malware properly adjusts the permissions (i.e. adhering to $W \oplus X$), only *Quincy* detected such injections. On Windows 7, we encountered the greatest difference in terms of ROC AUC score between *Quincy* and *Malfind*. It was almost nine percent. This is due to the fact that we detected many more malicious samples than *Malfind*. The increase was more than 27%. However, our system only decreased false positives by six percent. While there was a great variety of malicious samples in the training set, there was not enough variety of benign samples in it, which probably led to these false positives. On Windows 10, *Quincy* still dominated *Malfind*. It increased true positives by more than 13% but reached almost the same amount of false positives. This was caused by the highly skewed data set of Windows 10 (see Section 7.3.1). We assume that a better ratio between the malicious and benign samples will further increase the performance of our system (see also future work in Section 8.2.3).



(A) Windows XP

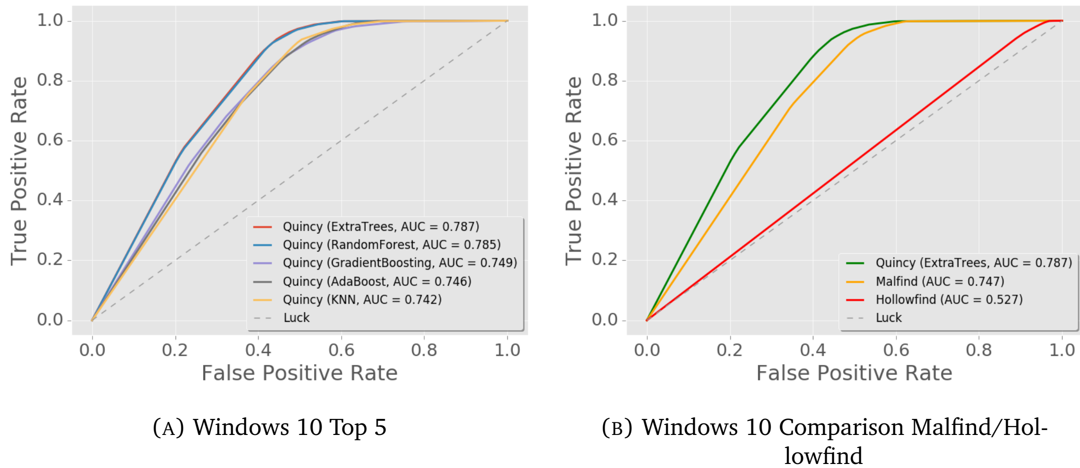
(B) Windows 7

FIGURE 7.7: ROC curves of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on Windows XP (left) and Windows 7 (right)

False positives included programs like *Dropbox Portable* that exhibits properties similar to malicious programs: high entropy areas due to compressed data, shellcode usage to determine the current position in memory, *RWX* memory region permissions, and cryptographic constants. *Dropbox Portable* may utilize these techniques to protect its intellectual property. There are two possible ways to deal with such false positives. First, continuous investigation of new features that allow for a better separation between goodware, malware, and goodware that is protected. Recently, Kim et al. [299] investigated this problem. Second, creating a whitelist of known goodware. This approach would be similar to *Hashtest* [26].

False negatives included malware like *Sakula* and *Rokku*. *Sakula* is a China-based RAT that is associated with zero-day exploits and good operational security [300]. It employs basic cryptography based on one-byte XORs that does not need to embed additional cryptographic constants. *Rokku* is a ransomware that employs *DLL* injections [301]. We only ran it on Windows 10. It encrypts strings on the fly and hence nulls our features that rely on strings. Even though there are other features that make it suspicious, the models of Windows 10 seemed to focus too much on memory region permissions, leaving out many *DLL* injections. As a consequence, future work should especially focus on DLL injections.

This brings us to another important aspect of combating false negatives: They greatly depend on the quality of the training data. Those splits that contained almost no *DLL*-based injection examples in the training set, showed a lower detection of such injections in the validation set. One reason for this is that the machine learning algorithms focus too much on memory region permissions in such cases. Therefore, it is important to have



(A) Windows 10 Top 5

(B) Windows 10 Comparison Malfind/Hollowfind

FIGURE 7.8: ROC curves of top five classifiers on Windows 10 (left) and *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on Windows 10 (right)

a healthy mix of less sophisticated injections exhibiting poor memory region allocation and sophisticated *DLL* injections that blend more into the benign regions around them.

In addition to a system's detection capabilities of a family, it is important that the detection is complete. Assume that a family has n artifacts in memory. Artifacts are, for example, the main malware module and its plug-ins but also further code on the heap. A system *detects* a family if it detects at least one of the n artifacts. If a system detects all n artifacts then it *completely* detects a family. A complete detection is very valuable to analysts since they have all pieces including the main module of the malware for further analysis.

Table 7.12 summarizes the family detection and the detection completeness of the three approaches. Our system detected on average more families than *Malfind* and *Hollowfind*. Since *Hollowfind* focuses only on a subset of all *HBCIAs*, it only detected some families and none of them completely. The difference between the detection of *Quincy* and *Malfind* was only marginal on Windows 10. But often *Malfind* did not detect the main module of a malware family, only a low-hanging fruit such as an uncarefully allocated *RWX* memory region.

Our system is more likely to completely detect a family. This is especially true for the most widespread system Windows 7 [8], where our system completely detected on average 18.1 families whereas *Malfind* only detected 11.9 families. Table 7.13 lists Split 2 of the Windows 7 evaluation as an illustration. The results are colored as follows: red denotes an undetected family, yellow denotes a partially detected family, and green denotes a completely detected family. Most families had less than six artifacts. *Hollowfind* only detected one family completely, six of them partially, and did not detect the rest.

Windows	Families	Quincy		Malfind		Hollowfind	
		detection	complete	detection	complete	detection	complete
XP	29	24.6	19.6	23.8	18.2	4.3	0
7	29	26.4	17.7	25	11.7	6.4	0
10	29	23.8	17.8	23.7	15.6	4	0

TABLE 7.12: Family detection and family completeness of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on $d_{validation}$.

Family	Total Artifacts	Quincy	Malfind	Hollowfind
Sinowal	2	1	2	1
Napolar	2	2	2	0
Backoff	3	2	2	1
Skynet	3	3	2	2
Sakula	2	0	0	0
Teerac	1	1	0	0
Feodo	5	5	0	0
Tempedreve	3	3	2	0
Phdet	4	2	0	0
Atrax	5	5	2	0
Symmi	3	2	1	0
Matsnu	1	1	1	0
Soraya	2	2	2	0
Citadel	36	6	4	0
Qakbot	7	7	7	0
Dorv	4	4	4	0
Tuscas	1	1	1	0
Tatanga	3	1	1	1
Vawtrack	5	4	4	0
Urausy	3	3	2	0
Tofsee	3	2	1	1
Rebhip	1	1	1	0
Spyeye	6	6	5	0
Kovter	5	4	3	2
Razorcrypt	1	1	1	1
Lethic	3	1	1	0
Gamker	5	5	4	0
Shiotob	1	1	1	0

TABLE 7.13: Example case of family detection and completeness: Split 2 of the Windows 7 evaluation. The table lists the results of *Quincy*, *Malfind*, and *Hollowfind*. Fully detected families are marked in green, partially detected in yellow, and undetected families in red.

Quincy detected more families completely than *Malfind* in this iteration. The ones that both systems partially detected had a higher coverage in our approach. *Quincy* missed only the malware *Sakula* as did every other system.

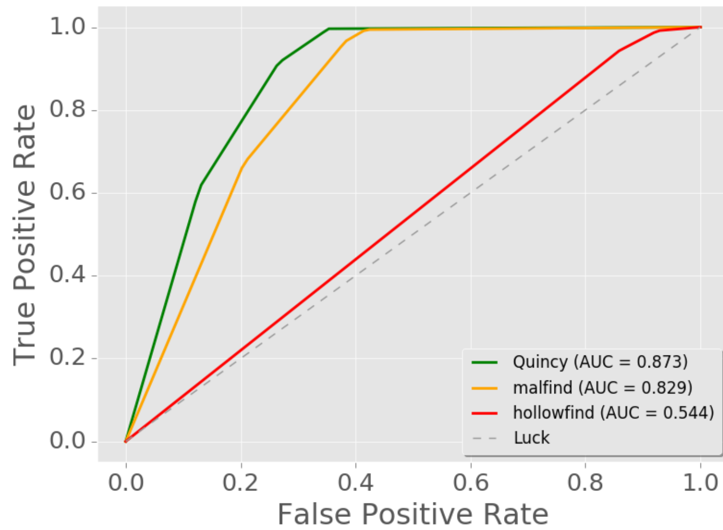


FIGURE 7.9: Averaged performance of *Quincy* (green), *Malfind* (orange), and *Hollowfind* (red) in temporal evaluation on Windows XP, Windows 7, and Windows 10 as ROC curve.

Temporal Evaluation

A machine learning classifier is trained on known, old data to predict on new, previously unseen data. We conducted a temporal evaluation to address this issue. The idea of a temporal evaluation was that a chronological order of the data was first established. Then, we trained on the older data and validated on the newer data. We queried the malware database *VirusTotal* [247] to establish the chronological order. *VirusTotal* stores the date of the first appearance of a sample. Then, we split the benign and malicious data sets into d_{train} and $d_{validation}$. We chose the same ratio as in the main evaluation in Section 7.3.2 (benign 1:9, malicious 6:4). Note that there was only one evaluation run per operating system since there was only one chronological order of the malicious samples.

The main result is that *Quincy* dominated *Malfind* and *Hollowfind* on all three operating systems. Figure 7.9 shows the averaged ROC curves of the three detection systems on Windows XP, 7, and 10. Table 7.11 summarizes the exact results. The results are in line with the main evaluation.

Another insight is that there are not two sets of old and new HBCIAs techniques. The same techniques can be found in both the older and newer samples, e.g. the classic injection technique employing *WriteProcessMemory* and *CreateRemoteThread*.

Windows	Number of Features	Number of Trees	Maximal Features
XP	28	50	$\sqrt{\|f\|}$
7	21	30	$\sqrt{\|f\|}$
10	17	82	$\sqrt{\|f\|}$

TABLE 7.14: Parameters of final *Extremely Randomized Trees* models. We created these models on the whole data set of each operating system. During a randomized grid search, two parameters were optimized (number of trees in ensemble and maximal features).

Final Models

The evaluation showed that the models produced by the algorithm *Extremely Randomized Trees* worked well on our problem. They provided a good trade off between true and false positives and they performed better than the other approaches. Therefore, we recommend utilizing *Extremely Randomized Trees* with *Quincy*. Note that its proof of concept implementation allows the user to create their own models and to freely choose the algorithm. Table 7.14 summarizes the number of features and optimal parameters of the final *Extremely Randomized Trees* models. They were created on the whole data sets of Windows XP, 7, and 10.

7.4 Conclusion

A fast initial detection of injected code in memory dumps is crucial due to the spread of HBCIAs. Not just forensic analysts but also malware analysts employ memory forensics due to the recent advances in frameworks like *Volatility*. Current HBCIA detection tools suffer many drawbacks such as high false positive rates that yield wasted time on the analysts side.

In this chapter, we presented *Quincy*, our system to detect HBCIAs in memory dumps. At its heart, it employs a machine learning heuristic utilizing up to 36 features. It discards invaluable features to improve detection performance and trains tree-based models for classification. The evaluation showed that Extremely Randomized Trees performed best in the realm of HBCIA detection. Due to our system's high amount of features, we assume that an evasion is significantly harder than in current systems.

We carried out our evaluation on a high quality data set, built according to current standards in malware experiments. Note that we published the data set on the code portal *github* [30]. The evaluation results showed that *Quincy* outperformed *Malfind* and *Hollowfind* on Windows XP, 7, and 10. While lowering the false positive rate, it increased

the true positive rate, detecting many advanced malware families such as *Dridex* and *Teerac*. It performed especially well on Windows 7, which is the most widely distributed version at the time of writing [8]. We published *Quincy* on the code portal *github* [29]. Whereas the publication enables researchers to build on our findings, practitioners can directly benefit from them and add our system to their work flow.

Wer schreibt, der bleibt.

8

Applications, Future Work, and Conclusion

In this final chapter, we conclude our thesis on the *Formalization and Detection of Host-Based Code Injection Attacks in the Context of Malware*. Firstly, we discuss applications of our work. Then, we continue to elaborate on future work that may lead to new and interesting topics. Finally, we conclude this thesis by discussing how we addressed the research questions that we posed in Chapter 1.

8.1 Applications

We expand on which areas our research and its results have a direct impact and how it is applicable in these areas. These areas are *Malware Detection*, *Malware Analysis*, and *Forensic Analysis*.

8.1.1 Malware Detection

First of all, our findings aid the detection of malware in general. According to our findings, almost two thirds of current Windows malware employs HBCIAs (see Section 5.1). However, HBCIAs are not restricted to Windows. Other operating systems like Linux,

macOS, or Android are also vulnerable to this kind of attack and malware already employs HBCIAs on these platforms (see Chapter 4).

In this thesis, we have presented *Bee Master*, an approach to detect HBCIAs at runtime. In contrast to traditional signature-based antivirus solutions, we detect malware based on this popular technique, *Host-Based Code Injection Attacks*. *Bee Master* can be deployed additionally to traditional solutions in order to improve the detection of malware.

8.1.2 Malware Analysis

Malware analysts face HBCIAs on a daily basis. Typically, they face the problem of finding the payload that has been injected in order to proceed with further analysis steps. This is very time consuming. Therefore, finding the payload right after injection saves analysts a lot of time and lets them focus on the main functionalities of the malware family. Both *Bee Master* and *Quincy* aid malware analysts during their daily work. Malware analysts have different preferences on how they tackle malware analysis.

Those who prefer dynamic analysis employ *Bee Master* to catch malicious payloads. We have shown (see Section 6.2) that already a small configuration with well-known victim processes (see Section 5.2) has a high probability of success. Those who prefer forensic analysis to receive a first overview of what happens on the system employ *Quincy*. It extracts malicious payloads reliably with only a few false positives. In addition, we published it on the code portal github [29] to promote open source research as well as to provide the malware analysis community a tool to detect advanced HBCIAs. As of February 2018, *Quincy* has 52 stars on github. Users star a project if they find it interesting and useful [302].

8.1.3 Forensic Analysis

Another field that directly profits from our findings is forensic analysis. We presented *Quincy* in Chapter 7. *Quincy* detects HBCIAs in memory dumps. It solves a daily task of forensic analysts. As we have shown in our evaluation of *Quincy* (see Section 7.3), it outperformed the current state of the art *Volatility's Malfind* as well as *Hollowfind*.

8.2 Future Work

Whereas this thesis gives an introduction to HBCIAs in the context of malware and presents two approaches for – statically as well as dynamically – detecting HBCIAs, there is still a lot more left to explore. We discuss future work in the following.

8.2.1 HBCIAs in General

We have measured the prevalence of HBCIA malware on Windows and have shown that it is a frequent phenomenon. Motivated by that, we have formalized HBCIAs and have derived a taxonomy of HBCIA algorithms. We are confident that based on our basic research, fellow researchers can carry out future research on HBCIAs.

Longitudinal Studies on HBCIAs: We conducted a longitudinal study on HBCIA prevalence in Section 5.1. This allowed us to measure the problem size on Windows. On the one hand, we would like to measure this on an even bigger data set like the ones that antivirus companies own to confirm our measurement. For instance, Panda labs collected 84 million unique malware samples just in 2015 [3]. These data sets would allow us to further approximate the problem size. On the other hand, such longitudinal studies should also be continuously carried out to judge the severeness of the problem and to observe any increase or decrease of it. Furthermore, other operating systems should also be taken into account, given their diversification.

Investigation of Future HBCIAs: We drew a comprehensive picture of today's HBCIAs in Chapter 4. We also provided an outlook of how future HBCIA-employing malware could look in Section 4.4. Future work should monitor and investigate new techniques for locally injecting code. There are continuously new HBCIA techniques like *Atom Bombing* that are already utilized in the wild [303]. Consequently they should be classified by our HBCIA taxonomy. Even though we claim that our taxonomy allows the classification of all current HBCIAs, future occurrences might demand adjustments to it.

8.2.2 Bee Master

We presented *Bee Master* to dynamically detect *HBCIAs* in Chapter 6. We have implemented and have evaluated it with several current malware families as well as artificial samples on several Linux and Windows versions.

Increase of Privilege Level: We have implemented a prototype of *Bee Master* in user space. *Bee Master*'s user space implementation comprises some drawbacks, which we

discussed in Section 6.1.3.1. In the same section, we also discussed possible implementations of the *Queen Bee* in kernel space or as virtual machine introspection component. Future work should focus on the reimplemention of the *Queen Bee* with higher privileges so as to it increases its stealthiness and tamper resistance.

Dynamic Worker Bees: *Bee Master* could go one step further and offer processes on demand. Malware often utilizes the same pattern for finding a certain process. It enumerates all running processes and compares, for instance, the process names to an internal list of names. If there is a match, it injects code into into the matched process. *Bee Master* could hook string comparison library functions like *strstr*. This would allow it to find the processes that the malware wishes to attack. These processes could be dynamically created and offered to the malware. This would, on one side, increase the chance of a successful attack on a *Worker Bee*. On the other side, this would increase performance since fewer *Worker Bees* would have to run at the same time.

Increase in the Number of Target Operating Systems: We intended *Bee Master* to be OS-agnostic, which we presented in its evaluation on Windows and Linux (see Chapter 6.2). However, there are more OSes than Windows and Linux that are vulnerable to HBCIAs. These include macOS and Android (see Section 4.1). Since the user bases of these OSes are steadily increasing, they also become a target of (HBCIA-employing) malware more frequently. Therefore, future work should port *Bee Master* to other operating systems like macOS and Android.

8.2.3 Quincy

We proposed our method *Quincy* to statically detect HBCIAs in memory dumps in Chapter 7. We implemented it for Windows XP, 7, and 10. Its evaluation with seven machine learning algorithms on a representative set of malware families and benign programs has shown its superiority over the current-state-of-the-art *Malfind*.

Exploration of new Features: Our system is already based on up to 36 features. This allows generic detection of many HBCIA-employing malware families and it also raises the bar for our adversaries to circumvent our approach (see Section 7.2.6). Better features would allow us to simplify the model and to improve its performance. Therefore, future work should explore new features. We have engineered the current feature set based on our domain knowledge, which is a common practice in machine learning [82]. This could be done semi-automatically. Possible information that the memory forensic framework *Volatility* [25] extracts should be surveyed first. Information that seems relevant could be added as a feature to Quincy, which extracts these features and only selects the relevant ones. Another line of research was proposed by Zhu et al. [304].

They extract features from other scientific publication by employing natural language processing. This would not yield completely novel features but features that are new to us. They may still be of great value to our system in combination with the current features.

Evaluation of Deep Learning Algorithms: We had to make a choice regarding the machine learning algorithms that we evaluated *Quincy* with. These were five tree-based algorithms, *KNN*, and *SVM*. We also mentioned that there is a plethora of other (supervised and unsupervised) machine learning algorithms.

Throughout the last decade, the term *Deep Learning* has received plenty of media attention. The idea behind *Deep Learning* is several decades old: *Artificial Neural Networks* (ANNs) [305]. ANNs were inspired by biological neural networks found, for instance, in brains of humans [305]. They consist of connected units (resembling biological neurons) that transmit signals. These units are organized in different layers with at least one input and one output layer. There may be several hidden layers. ANNs with several hidden layers are also denoted as *Deep Neural Networks* (DNNs) [305].

However, the more hidden layer a DNN contains, the higher the training cost. Since the mid 2010s, graphics processing units (GPUs) have been utilized to train DNNs, which are well suited for massively parallel matrix computations required by DNN algorithms [306]. Today, global players such as *Baidu* [307], *Facebook* [308], and *Google* [309] invest millions of dollars in DNNs, for instance, producing state-of-the-art artificial intelligence applications in the fields of computer vision and voice recognition.

DNNs were also applied to the domain of malware classification and detection, for instance, Saxe et al. [310] (2015), Kolosnjaji et al. [311] (2016), and Kolosnjaji et al. [312] (2017). The results of these papers showed astonishing results, for example, the detection rate of Saxe et al. [310] reached 95% on malicious binaries. While the aforementioned approaches utilized DNNs, they still comprised a feature engineering phase, where expert domain knowledge was required to create the features. These papers have shown that DNNs are applicable to the domain of malware classification and detection.

The advances of DNNs are also due to the fact that huge data sets are available to train the data hungry DNNs [305]. This is also the case for static malware analysis, where one only has to aggregate binaries [310]. However, our problem is different. The memory image of a malware usually differs ignificantly from its image on hard disk. Therefore, a first prerequisite to apply DNNs to our problem is even more training data. We pick up this point later on in this section.

Nevertheless, we believe that DNNs are a promising research branch in order to further increase the detection performance of *Quincy*. Consequently, future work should evaluate *Quincy* with *Deep Learning* algorithms. There are still several open questions like *Can we apply automatic feature extraction to our domain in order to forego expert domain knowledge?*, that future work will address.

Evaluation of Stacking Algorithms: Another path to follow would be *Stacking*. This is a form of ensemble learning and it trains a machine learning algorithm that combines the prediction of several other machine learning algorithms [313]. A *Stacking* ensemble, for instance, could utilize a *Gaussian SVM*, a *Random Forest*, and *KNN* as base classifiers and could train *Logistic Regression* on the base classifier's predictions. *Stacking* is often employed in machine learning challenges, for example, the *Netflix Prize* competition was won with a *Stacking* model [314].

Collection of more Training Data: The quality of a machine learning model greatly depends on the quality of the training data. We have argued that while it is easy to collect properly labeled benign data, it is tedious to collect properly labeled malicious data (see Section 7.3.1). This was the reason why the class distribution between benign and malicious samples was skewed. However, the more malicious data that is available, the better the machine learning algorithms can build models that detect HBCIA-employing malware with high accuracy. Future work focuses on the continuous extension of properly labeled malicious training data.

Recently, Plohmann started the *Malpedia* project [315] that aims at creating a comprehensive high quality malware collection. We contributed the data set presented in Chapter 7 to this malware collection. The project is a community approach and allows malware families to be tagged. If we could tag more families that employ code injections then this would lead to more evaluation data to improve *Quincy*. Future work will focus on that and continue to gather high quality data to train our system.

Increase in the Number of Target Operating Systems: *Quincy* utilizes features that are not restricted to one operating system. The current implementation works on several versions of Microsoft Windows. However, HBCIAs are an OS-independent problem and they can be found on other OSes like Linux and macOS. Future work deals with porting *Quincy* to other OSes. It employs the memory forensic framework *Volatility* to analyze memory dumps. *Volatility* serves as an abstraction layer of the dump's underlying operating system. The framework implements support for Windows, Linux, Android, and macOS [66]. Therefore, increasing *Quincy*'s number of target operating systems would be a matter of implementation work.

8.3 Conclusion

This thesis has focused on *Host-Based Code Injection Attacks* (HBCIAs) in the context of malware. HBCIAs allow malware to execute its payload within another process space without requiring it to provide its own. We posed two research questions in Chapter 1 that set our path for investigating HBCIAs. In the following, we discuss these two research questions again and see how we have addressed them. The first question aimed at HBCIAs in general.

> How does modern malware employ Host-Based Code Injection Attacks, what are the consequences, and what is the problem size?

There are many reasons why malware authors utilize HBCIAs including detection evasion and interception of information (see Section 4.2). HBCIAs are popular among current malware as we have shown, roughly two thirds of malware employs them so that they are a relevant problem to security researchers (see Section 5.1). We have built a foundation for future research on this problem by conducting basic research and problem formalization. The detection of HBCIAs implies the detection of a plethora of malware families. On one hand, this includes widespread cybercrime families such as *Andromeda*, *Nymaim*, and *Zeus*. On the other hand, this includes advanced persistent threats like *Duqu*, *Flame*, and *Stuxnet*. Furthermore, the detection of HBCIAs could possibly be utilized to detect unknown malware families. Therefore, we have focused on the (dynamic and static) detection of HBCIAs in our thesis.

The second research question focused on the detection of HBCIAs in two different settings.

> How can we detect this malicious behavior using dynamic and static techniques?

We have intensively researched this topic. The outcome is two approaches to detect HBCIAs in the context of malware.

Our first method *Bee Master* detects HBCIAs dynamically and platform-independently. We have achieved this by applying the honeypot paradigm to OS processes. We have shown in our evaluation on Microsoft Windows and Ubuntu Linux that it is feasible without having false positives. We believe that if *Bee Master* were deployed alongside traditional antivirus software that this would significantly increase end user security.

Our second approach *Quincy* statically detects HBCIAs in memory dumps. It considers up to 36 features to detect code-injecting malware. Internally it utilizes a tree-based

machine learning decision heuristic. The machine learning algorithm is replaceable, we have discovered that *Extremely Randomized Trees* worked well on our problem. We have carried out an evaluation of more than a thousand benign and malicious memory dumps of three Microsoft Windows versions from Windows XP to Windows 10. The results have shown that Quincy is superior to the current state of the art *Malfind*. It increased the true positive rate while significantly decreasing the false positive rate.

Sample List of Class Prevalence Estimation

All malware families used in the prevalence estimation of the different HBCIA algorithms (see Section 5.4) are listed in Table A.1. We use the labels provided by Microsoft Security Essentials [316] in order to be consistent.

Malware family	MD5 hash
Afcore	09af9958262de0066faabfc844a42137
Alureon	32d6644c5ea66e390070d3dc3401e54b
Bebloh	c08bae454dcedd26ab529de721ea4460
Caphaw	dcc876357354acaf2b61ee3e839154ad
Carberp	02e693a4a66c104541dec55f417d29b9
Citadel	97e545aae517a5f816abcd960875ac05
Conficker	ac4f94c0b774b9a7b85582724c48275f
Cridex	734aadd62d0662256a65510271d40048
Conhook	b8ce960f658b03e9147fc3ed700dfbd3
Cutwail	0c699bf8815137404fc43f6e56761ac8
Dofoil	ac576a1983f6c455bf6a5e4f587ebc3a
Dorkbot	393b4c117e15fbcfe56f560a8e6a3f0c
Duqu	9749d38ae9b9ddd81b50aad679ee87ec
Enfal	fef64fa89841d276b0c8bf9417f398b6
Eyestye	34bd32ff879c86b48e8eaf4d0cfebc8c
Gamarue	c2d7977da17a4e22b2c4a9d40f7ff51a
Gataka	576f95b855f69981cace04eb9ff22e11
Hesperbot	2323e06279e8b9f93a427bacfba3b953
Kuluoz	f48a6740f2e0d70343f600be12220adc
Lethic	0460d89f0091d951184a8d77c6641340
Matsnu	ffcf2bb69f23c7c234d2f2ee380cdaa4
Napolar	7d6fbfe63c5c126ed585880b54844edd
Neurevt	d9b1b19d2df8f489e38297ea89059900
Nymaim	bb4bf5036ee175ac09b16e0989c441fa
Poison	001b8f696b6576798517168cd0a0fb44
Quadars	8cee78fcd2d5f98914ce38d2035a3d02
Ramnit	49e486fcc7da44f12a4598258011b580
Redyms	0044d66e4abf7c4af6b5d207065320f7
Rombrast	06cdd36673a29822360907f8abec6a59
Rovnix	4244dba80a00da7e6c581b4d2ced3277
Sality	63c27ea5328f862ac1ff1674177a189c
Screud	089c5446291c9145ad8ac6c1cdfe4928
Seedna	5024ce13efab0e531c4e09b98def1287
Sirefef	c6e73a75284507a41da8bef0db342400
Stuxnet	ec591cf5f3b915813e14060d3057395b
Trxa	96fecadd17682ce64e68887f018e12e3
Ursnif	5622d3aab0ecf0f229b47005e306f49e
Virut	021f641bb2485cb25dcc6de2107e915c
Vundo	aaa7a750adefcff4059a140fd69795bb
Zbot	3cfc97f88e7b24d3ceecd4ba7054e138

TABLE A.1: Malware families considered in HBCIA algorithm prevalence estimation

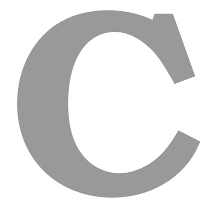
B

Sample List of Bee Master Evaluation

This chapter contains information about the malware samples considered in the evaluation of *Bee Master* (see Chapter 5). All malware families are listed in Table B.1. In case we had to pick more than one representative due to compatibility reasons or we could not execute any sample of the mentioned family we explicitly state which sample was used on what platform. Otherwise the listed sample worked on all platforms. Furthermore, we utilized the labels provided by Microsoft Security Essentials [316] in order to be consistent. Unfortunately, there were few families where we could not determine a Microsoft specific label. These cases are explicitly marked.

Malware family	MD5 hash
Bamital	Windows XP/7: b72eae1db843005fb303dc96c4e98593
Bebloh	79ba32519af63486facaa262d88ee4ea
Caphaw	9466be80af54640218fce4351cc19a41
Carberp	02e693a4a66c104541dec55f417d29b9
Citadel	699e84682acdf3304fc79014e30eb11f
Conficker	Windows XP/7: 8c9367b7dc43dadaa3ec9da767c586cf
Cridex	3d919e36029cc92724a7b9915abd8075
Dorkbot	74c7725c2df337cf860990660d63520f
Eyestye	Windows XP/7: 34bd32ff879c86b48e8eaf4d0cfebc8c
	Windows 8: 02ea29c0b04725f9b9936129de127133
Feodo[1]	557597074df3d3ce0e1674285ef19732
Foidan	991327984cc14474ad4e863a2543bad9
Gamker	Windows XP/7: c9197f34d616b46074509b4827c85675
Gapz	089c5446291c9145ad8ac6c1cdfe4928
Gataka	8d000237aeb45310951185ea895d85ed
Hanthie	Linux: 761b6266c5254513bc1509d0a36becbd
Hesperbot	12bb85fafd5826fb988c2bee03175632
Ice IX	Windows XP/7: e661ff3d8ae16ab40b8638b8a74fff2b
Lolyda	Windows XP: 6f9fc55fc5323704d464794b25dc8a56
	Windows 7/8: 235b650a98740db60fefe05a427eda74
Napolar	5418869ee4700f3893ba067109b3bd2e
Neurevt	48048cfbf579c73b9587333d8768c282
Nymaim	628ba5d2ed6ca6df41863057641047ae
Parite	3689dd289c6c00cc6586bb354f8f2530
Poison	Windows XP: 1ab647cd1c08e542d0cf922b3c8432d0
Ramnit	72a792e3d044dbe3db66971501c268b1
Redyms	0044d66e4abf7c4af6b5d207065320f7
Rombrast	Windows XP: dcc8d837dbb6cbfdb49270acf9274e3f
	Windows 7/8: 33edb276c62afe4aba1f4f1907818135
Sality	5e4f1f1aa595c354413090e172e8fd91
Sazoora	dce968bdae6f1a0ee29046e439b24cd6
Shiotob	01cc26be43086375ff6e6f95318b78b0
Sinowal	007799fc41bc1fb39ff8cff8cb3478b2
Skynet	191b26bafdf58397088c88a1b3bac5a6
Sykipot	Windows XP/7: 4960dd192384469129f0a1bcd2b5ae83
Tinba	6244604b4fe75b652c05a217ac90eeac
Trxa	2a41db8710f165c98f5717818ff3d7be
Ursnif	01ab2ed9e551a9a40c426a826a5a0c9b
Vawtrak	7b05cc5f48c389a53a42ca1a8e4b2957
Virut	ef15ddfb52c443dd2e4698115c4f1a69
Zbot	Windows XP/7: ab0587cd3872e14e3dfbf2503a34e42c
	Windows 8: 0c5df80b23b7712bc39655d79549b0b4
ZeusP2P	203f031a7d41fb247d0bd55bb8b1f382

TABLE B.1: Representatives of malware families



Sample List of Quincy Evaluation

Evaluated Malware Families

The tables C.1, C.2, and C.3 lists all malware families that we evaluated *Quincy* with (see Section 7.3) on the three considered operating systems Windows XP, Windows 7, and Windows 10.

Selected Features of Final Models

Table C.4 lists the features that were selected for the final models of *Quincy* on Windows XP, Windows 7, and Windows 10.

Malware Family	SHA256 Hash
atrax	0bc2db99f5277ddc89409a8c487298df5dbf34108146c58885ad0c804422c27f
betabot	0f79f1c07e299f32eb788b6f710926f072968923198b695eaa592191232d15c3
bugat	423f80b92bb6d9dc200d897e927085f06790582055df7a9059c6c33ea8968e7a
carberp	a06cd8879a3e6f69b0f87d6aca0fd568de02f8ff39c9e2ee987f506bc76ec447
citadel	671a29ffd716e52ed3be9702de65c1be526b2dff30775e60c1a234753da44918
confickerB	16fb3685a888668982f98dc7c37db9ab32fd0cab5d9957ea9fa535d1cc1152fb
conhook	bfbeaf964ec0a557b97aa5c3fceb179c328fa7f63ff0e8b28ce60f8a27f4122d
cryptolocker_racketeer	5ace9d2d71d991ea6b74531f42e5bf58866be71cba203494788f9fe3ccfd4878
dalixi	464d963f698ee0a385a983c5b88b1c2ba3243ebb776e8f205def48d7aac348d7
darkcomet	8a18ea64a05e6c56024beebc832b65e83337be594dfd4337f42b057e2401f660
dofoil	9cc683c5f44c97f02e10a9df34296e9ba58f7446d954d122c22b02f3dc21a55e
dorkbot	952f756224303dcca8328a8c774d935246d28a901b554990eb2a521c407d6f8e
dorv	97501ced92451792dcb5bdc82fb775e44a272f93f24367ede7e3e421babd2c4e
enfal	370b53e30c026bcb05549791734755fc9de58072ce709724b74f2597518f43a2
feodo	e544337e67b9a50824b2f124133abab3d213c3c3e15736f50c156d78638dfdf3
foidan	077dcc584462e400cb502c4507d301b27213851945ed0e494d9d55036b1f5a1e
gamker	2739511e2efc846c144118ebf524b16d2127dd8f2128d909c5095bfe5272560c
gapz	19d1aaef16cf892bd8e0ea37fff29feeb540fd122b288b7aae4a4212a2dbd93b
gataka	3eaaeea3e4e649492e5c98d48af6b925d52795eb64c081d0e62b9cc0df5c9f29
hamweq	1f526ff74d22aff9597d3ba341b0a90ae4df45616776ee1c7175ab3551557b25
isfb	0e61ecd0aad87a72d36bc10288303292859a800d2237ac9c32755d9e455e87e2
kins	464b5d92bdfee3eecf1af64618aa1783cde470462a301b21780bb92e2e0ded96
kovter	c9444bf21cc0b709e8a1a434441ebc5be4037a63617585298f0137e820e64ddb
kuluoz	f7ca17ceb289db28aae81106e2dde9571f0ea43249f78a38622b582397029109
lamechi	654dd74621ea41b4809d09c18bc3a9d8934eed9ccd20749171ba45f63d042961
lethic	15f4622d0ea4f6a5afbe5ef725bd5e289290116b2d5613f91e733d77f27a8c0a
liftoh	8a4a8ffad419e2a39a20b2c491f59d54b9ba014dcf7671552bd34fec90649300
matsnu2016	d21f6e2fc7bd528240b3f5215d2015a0e290cc0c828487193a1fa2d0a9197ccf
napolar	760c76455a6271b53afdb40678d2dca2e4339d4df6ac5ac24ec4e2403305baa9
necurs	bc56597decc0815b11f47ef6882f07ab0e570764ebe77192841968cfc44c92ea
nymaim	d9be07e6f40a8c77cd2c36d979e6079ce5756b7923f1040cad095744fc402031
parite	50f1fd9e191f99e7bbd77bfc1714036fd537a607bb6375315c06186c79e89adb
phase	1e628c36d22a9b8e493901bb6c5e17acb1bfcf0dea65f4012c39e6f9bc827a9c
phdet	ca7a8180996a98e718f427837f9d52453b78d0a307e06e1866db4d4ce969d525
poisonivy	ca4f35a7a6f98cbd9e065a171675f628c317f4365c01911f10160fd8bed87b1c
qakbot	057801414d1ca1785c3f87569878673bd97ffb3e189fc58a1155207110c9a17c
ramnit	b2b56ff4227034bcb2d537c98c41df8be94b7ac58bcbd11f8bc7b46c3ebc5ca5
razorcrypt_normal	fa0b3b18d33d388c3d71392eb7a598f73e1ac4b38086a96f60170c7aac081506
razorcrypt_notepad	97e0eb1145eb284ab5e0f3cea77a3e067f6ed4a7e5fce4967bf6d70df5cc92b0
rebhip	d7ace08c2005104ae1d88a0a5b0752112c3488cc38431f50e8b937eef906f9ed
redyms	b6d19c3e6e82bbde62984f50144ce4d98a18871374ec5d313489d5831317c480
remtasu	7f9ae2fb8e0a2ef18ab16afcb5053e425a42c0caa8080210707893dfdf328936
rombrast	1e87d2cbc136d9695b59e67f37035a45a9ad30f5fccc216387a03c0a62afa9d4
sality	4483287e984256e44e9c677080170758ae76bc2addc1d1de8f0ea468af485623
savnut	ae836113c424ef36efcfdc6c8493f466a3bd31cfb4b75f6cf26b0d715a58d06b
schwarzesonne	aa37ece103da4a5c94079b66d13d2112c908094fa3dfd70b89dc07b198f92985
sefnit	3b2e246d5460d214ad07d2b395b95b77e0679f46dc563e6eb2ff067c650fad8c
shiotob	c8ed42a6791ef4dd207ac6f0a481edecb32caf59e6b5c058b59f5847c7f1bd0a
shylock	35ccf0e051fb0ff61e52ff4130eb38521f954f90ad9c97be59af1e7901974557
simda	9be21e762aa084c362ac630be9f5971e9655dc7016335c68f7e83defe11f7344
skynet	880ce96c128e416c30e9786f5b453030137ee2ef9ee9abf32267d3d6444c4119
sopinar	f711606b253aa4faf3c1e0f9b8f6bbb9e246ce1a0320ab779833bbc166d63ac1
soraya	c1a2a7bbd65d61a6ecd77fe9e921e39fad770a8cbcf15ea7ba0cb9066ce1656c
spyeye	e807eedad5ad83238de5c6ee9ff3cbed0e9599ce94dc7e03581960ab7c665709
symmi	8f98246c03a8e6bf2c342cb5ebc9e82e0cb596c333e5cab0eaed2dc247378755
tatanga	224e3fa69a82f2b9ac5b53d1e0b5c6a339ef5add0b8b2bd5fc9d2ac706ecf2e4
teerac	447fe9c756081e328f5855b60f3f5c1af460c6d86df0bc671556c60515531fa1
tempedreve	77f24ca7bbbfdcd4d182d794e19243ada9e97e60e36ad091797c500e5c4de6af
tinba	09478bf4833505d3d7b66d4f30ccce6b9fde3ea51b9ccf6fdeadc008efba43d8
tinybanker	df4aa8c80b7f2c091eee11952b30d5ed2638dd311c9120d4e742db41805a34de
tofsee	a8c8849605a0eebaa06202dc763f47e10c668e46d53f8d96562d59ee52b6b801
trustezeb	1f8992bb765566cce72dfc77c9f54916480d87aa70b88994333578d23c06e1dc
tuscas	9ef9d58f869f6b737383fb3b659e48910a7593d720e9f6479d618d396b5ddc56
urausy	e4f821e0c469ef6030f1b309a56a313f25a88e4f91a3ef28387785bb07ebcd8e
urlzone	a639e0d31e44fbb6786733325da5938b8086877f893b6d1cdb78d40e50d32a4b
vawtrack2016	b6260c3f504ebab55a1c13bfdc88c04ea20265a661f3445a6c040c91556cbf4f
vawtrak	3547187819ee0ab9c72530dc5dbe79b81c84dbc4a57a274e0044ce997fb6982b
vundo	3563ce2be8aa67950875c8b0678ed79636bbe7e26215f8a25f32e316a70a6e25
xswkit	34f9e77c56549fa25b02ae97891054b963b4bc9e32971b65cc6f203e8a2792f1
zeroaccess	be00ef4081af7d101b36e4552e6ca7ba2cd91f906cc5dff8751ef2078026a774
zeusp2p	4bbf0a795cf4d66af449723332c94c9c4ba0edd7ae23809259bab82375b7da81
zurgop	d12526fc430fa213d77f8523a89c92c5f4e0d11deacbaf5c160a16f87ed5adc3

TABLE C.1: Malware families of Quincy's Windows XP evaluation

Malware Family	SHA256 Hash
atrax	0bc2db99f5277ddc89409a8c487298df5dbf34108146c58885ad0c804422c27f
backoff	bf693160f8a2857ba9f16ca028e6a63074c0a3038aa3dee7992f37e04c04469e
bamital	28da9324aa6b9e9537c51ac6a2b188c3a3c1d41eeaf6f7d15b6630794d8cd8cc
betabot	0f79f1c07e299f32eb788b6f710926f072968923198b695eaa592191232d15c3
carberp	a06cd8879a3e6f69b0f87d6aca0fd568de02f8ff39c9e2ee987f506bc76ec447
citadel	671a29ffd716e52ed3be9702de65c1be526b2dff30775e60c1a234753da44918
corebot	161cdb6b51702ca0d065808e7f36590114134923ddb4d459f8f37b68bf284eff
cryptolocker_racketeer	5ace9d2d71d991ea6b74531f42e5bf58866be71cba203494788f9fe3ccfd4878
cryptowall	cc83b186700b21e5c4cae0f8236ae3e50ab47c2c21a3987ea00463056cbd1c26
darkcomet	8a18ea64a05e6c56024beebc832b65e83337be594dfd4337f42b057e2401f660
dofoil	9cc683c5f44c97f02e10a9df34296e9ba58f7446d954d122c22b02f3dc21a55e
dorkbot	952f756224303dcca8328a8c774d935246d28a901b554990eb2a521c407d6f8e
dorv	97501ced92451792dcb5bdc82fb775e44a272f93f24367ede7e3e421babd2c4e
enfal	370b53e30c026bcb05549791734755fc9de58072ce709724b74f2597518f43a2
feodo	e544337e67b9a50824b2f124133abab3d213c3c3e15736f50c156d78638dfdf3
foidan	077dcc584462e400cb502c4507d301b27213851945ed0e494d9d55036b1f5a1e
gamker	2739511e2efc846c144118ebf524b16d2127dd8f2128d909c5095bfe5272560c
gapz	19d1aaef16cf892bd8e0ea37fff29feeb540fd122b288b7aae4a4212a2dbd93b
gataka	3eaaeea3e4e649492e5c98d48af6b925d52795eb64c081d0e62b9cc0df5c9f29
hamweq	1f526ff74d22aff9597d3ba341b0a90ae4df45616776ee1c7175ab3551557b25
isfb	0e61ecd0aad87a72d36bc10288303292859a800d2237ac9c32755d9e455e87e2
kins	464b5d92bdfee3eecf1af64618aa1783cde470462a301b21780bb92e2e0ded96
kovter	c9444bf21cc0b709e8a1a434441ebc5be4037a63617585298f0137e820e64ddb
kuluoz	f7ca17ceb289db28aae81106e2dde9571f0ea43249f78a38622b582397029109
lethic	15f4622d0ea4f6a5afbe5ef725bd5e289290116b2d5613f91e733d77f27a8c0a
liftoh	8a4a8ffad419e2a39a20b2c491f59d54b9ba014dcf7671552bd34fec90649300
matsnu2016	d21f6e2fc7bd528240b3f5215d2015a0e290cc0c828487193a1fa2d0a9197ccf
napolar	760c76455a6271b53afdb40678d2dca2e4339d4df6ac5ac24ec4e2403305baa9
nymaim	d9be07e6f40a8c77cd2c36d979e6079ce5756b7923f1040cad095744fc402031
parite	50f1fd9e191f99e7bbd77bfc1714036fd537a607bb6375315c06186c79e89adb
phase	1e628c36d22a9b8e493901bb6c5e17acb1bfcf0dea65f4012c39e6f9bc827a9c
phdet	ca7a8180996a98e718f427837f9d52453b78d0a307e06e1866db4d4ce969d525
qakbot	057801414d1ca1785c3f87569878673bd97ffb3e189fc58a1155207110c9a17c
ramnit	b2b56ff4227034bcb2d537c98c41df8be94b7ac58bcbd11f8bc7b46c3ebc5ca5
razorcrypt_normal	fa0b3b18d33d388c3d71392eb7a598f73e1ac4b38086a96f60170c7aac081506
razorcrypt_notepad	97e0eb1145eb284ab5e0f3cea77a3e067f6ed4a7e5fce4967bf6d70df5cc92b0
rebhip	d7ace08c2005104ae1d88a0a5b0752112c3488cc38431f50e8b937eef906f9ed
redyms	b6d19c3e6e82bbde62984f50144ce4d98a18871374ec5d313489d5831317c480
remtasu	7f9ae2fb8e0a2ef18ab16afcb5053e425a42c0caa8080210707893dfdf328936
rombrast	1e87d2cbc136d9695b59e67f37035a45a9ad30f5fccc216387a03c0a62afa9d4
sakula	22f5fa60c2286e22bee79bcde6e9c7ee80b42ef308c6bb7aed6d6163e5da0214
sality	4483287e984256e44e9c677080170758ae76bc2addc1d1de8f0ea468af485623
sality2016	1d441a7c0f32513163b7e287cfa4869d61b750acf212449bc98193a30ea47d23
savnut	ae836113c424ef36efcfdc6c8493f466a3bd31cfb4b75f6cf26b0d715a58d06b
schwarzesonne	aa37ece103da4a5c94079b66d13d2112c908094fa3dfd70b89dc07b198f92985
shiotob	c8ed42a6791ef4dd207ac6f0a481edecb32caf59e6b5c058b59f5847c7f1bd0a
shylock	35ccf0e051fb0ff61e52ff4130eb38521f954f90ad9c97be59af1e7901974557
simda	9be21e762aa084c362ac630be9f5971e9655dc7016335c68f7e83defe11f7344
sinowal	c7a65cea20c1a312c3619ed5923c2d1eee55afc26f21f1ce9e01179aec0db3ad
skynet	880ce96c128e416c30e9786f5b453030137ee2ef9ee9abf32267d3d6444c4119
sopinar	f711606b253aa4faf3c1e0f9b8f6bbb9e246ce1a0320ab779833bbc166d63ac1
soraya	c1a2a7bbd65d61a6ecd77fe9e921e39fad770a8cbcf15ea7ba0cb9066ce1656c
spyeye	e807eedad5ad83238de5c6ee9ff3cbed0e9599ce94dc7e03581960ab7c665709
symmi	8f98246c03a8e6bf2c342cb5ebc9e82e0cb596c333e5cab0eaed2dc247378755
tatanga	224e3fa69a82f2b9ac5b53d1e0b5c6a339ef5add0b8b2bd5fc9d2ac706ecf2e4
teerac	447fe9c756081e328f5855b60f3f5c1af460c6d86df0bc671556c60515531fa1
tempedreve	77f24ca7bbbfdcd4d182d794e19243ada9e97e60e36ad091797c500e5c4de6af
tinba	09478bf4833505d3d7b66d4f30ccce6b9fde3ea51b9ccf6fdeadc008efba43d8
tinybanker	df4aa8c80b7f2c091eee11952b30d5ed2638dd311c9120d4e742db41805a34de
tofsee	a8c8849605a0eebaa06202dc763f47e10c668e46d53f8d96562d59ee52b6b801
trustezeb	1f8992bb765566cce72dfc77c9f54916480d87aa70b88994333578d23c06e1dc
tuscas	9ef9d58f869f6b737383fb3b659e48910a7593d720e9f6479d618d396b5ddc56
urausy	e4f821e0c469ef6030f1b309a56a313f25a88e4f91a3ef28387785bb07ebcd8e
urlzone	a639e0d31e44fbb6786733325da5938b8086877f893b6d1cdb78d40e50d32a4b
vawtrack2016	b6260c3f504ebab55a1c13bfdc88c04ea20265a661f3445a6c040c91556cbf4f
vawtrak	3547187819ee0ab9c72530dc5dbe79b81c84dbc4a57a274e0044ce997fb6982b
vundo	3563ce2be8aa67950875c8b0678ed79636bbe7e26215f8a25f32e316a70a6e25
xswkit	34f9e77c56549fa25b02ae97891054b963b4bc9e32971b65cc6f203e8a2792f1
zeroaccess	be00ef4081af7d101b36e4552e6ca7ba2cd91f906cc5dff8751ef2078026a774
zeusp2p	4bbf0a795cf4d66af449723332c94c9c4ba0edd7ae23809259bab82375b7da81
zurgop	d12526fc430fa213d77f8523a89c92c5f4e0d11deacbaf5c160a16f87ed5adc3

TABLE C.2: Malware families of Quincy’s Windows 7 evaluation

Malware Family	SHA256 Hash
afcore	f76700cbfc4176e0976a20660dafbf4e5e6d560a08249e1ed1f187ae8e4f2a6b
aleksandr	171693ab13668c6004a1e08b83c9877a55f150aaa6d8a624c3f8ffc712b22f0b
andromeda	76c65d79ade16f9b461d38f7e2433ac66c0569b36988705383f4b9cb64635af3
atrax	0bc2db99f5277ddc89409a8c487298df5dbf34108146c58885ad0c804422c27f
backoff	bf693160f8a2857ba9f16ca028e6a63074c0a3038aa3dee7992f37e04c04469e
bamital	28da9324aa6b9e9537c51ac6a2b188c3a3c1d41eeaf6f7d15b6630794d8cd8cc
citadel	671a29ffd716e52ed3be9702de65c1be526b2dff30775e60c1a234753da44918
confickerB	16fb3685a888668982f98dc7c37db9ab32fd0cab5d9957ea9fa535d1cc1152fb
corebot	161cdb6b51702ca0d065808e7f36590114134923ddb4d459f8f37b68bf284eff
cryptolocker_racketeer	5ace9d2d71d991ea6b74531f42e5bf58866be71cba203494788f9fe3ccfd4878
cryptowall	cc83b186700b21e5c4cae0f8236ae3e50ab47c2c21a3987ea00463056cbd1c26
cutwail	d9a21f5a7c8560ce9a1368943509f791b568d18c2abd329cbb095662a7642ed6
darkcomet	8a18ea64a05e6c56024beebc832b65e83337be594dfd4337f42b057e2401f660
dofoil	9cc683c5f44c97f02e10a9df34296e9ba58f7446d954d122c22b02f3dc21a55e
dorkbot	952f756224303dcca8328a8c774d935246d28a901b554990eb2a521c407d6f8e
emotet	284f9e5fae27f0a1646da804a762b8ede8a01f46bda184eb30fee02c16a77473
enfal	370b53e30c026bcb05549791734755fc9de58072ce709724b74f2597518f43a2
finfisher	f827c92fbe832db3f09f47fe0dcaafd89b40c7064ab90833a1f418f2d1e75e8e
fobber	5d0d798cf8ef16cc57ff57999f70f25d02c7c69a1dc67114dd5766c990771469
foidan	077dcc584462e400cb502c4507d301b27213851945ed0e494d9d55036b1f5a1e
gataka	3eaaeea3e4e649492e5c98d48af6b925d52795eb64c081d0e62b9cc0df5c9f29
gookit	a6c1d77964664f85259727acc647c56113c6a71adef94c0460bc49ae5ee49188
gozi	5da54366629021851ae47876bdffcc501c809f503777cd082fbec7b2148c5354
goznym	792c968786d7b161232a683f1b38beb83e26807d4e9741638291c46478dac206
isfb	0e61ecd0aad87a72d36bc10288303292859a800d2237ac9c32755d9e455e87e2
kins	464b5d92bdfee3eecf1af64618aa1783cde470462a301b21780bb92e2e0ded96
kovter	c9444bf21cc0b709e8a1a434441ebc5be4037a63617585298f0137e820e64ddb
kronos	792c968786d7b161232a683f1b38beb83e26807d4e9741638291c46478dac206
lethic	15f4622d0ea4f6a5afbe5ef725bd5e289290116b2d5613f91e733d77f27a8c0a
liftoh	8a4a8ffad419e2a39a20b2c491f59d54b9ba014dcf7671552bd34fec90649300
lusypos	d7a08338bcb30cc688a827b611fe9b26c54f3ba35c02355fa1d468da8cbbd903
matsnu	472dcbd07ff6679ecfb11ac924cac3f70bb56c24ea1de30da5652ceaacae3b7c
matsnu2016	d21f6e2fc7bd528240b3f5215d2015a0e290cc0c828487193a1fa2d0a9197ccf
necurs	bc56597decc0815b11f47ef6882f07ab0e570764ebe77192841968cfc44c92ea
pandabanker	1cccc844fcdb255f833a9ef36c2d3c690557b828ed5d0a45d068aeb2af1faac7
parite	50f1fd9e191f99e7bbd77bfc1714036fd537a607bb6375315c06186c79e89adb
phase	1e628c36d22a9b8e493901bb6c5e17acb1bfcf0dea65f4012c39e6f9bc827a9c
phdet	ca7a8180996a98e718f427837f9d52453b78d0a307e06e1866db4d4ce969d525
qadars	04f0feb7ab6b68dabbebc3da2082afabacbe488bca3464f3da985a9b4d9c7c2c
qakbot	057801414d1ca1785c3f87569878673bd97ffb3e189fc58a1155207110c9a17c
ramnit	e9cd4d4affcb97b298fafe52a1a7f8e054e0ea4ac9ce0f9dc0bc4a199a10e30f
ranbyus	dc4f3340ca8e623a5a77eb95411696fc25a7e6f5ef657ac9fd76eb4bc11c16b4
rebhip	d7ace08c2005104ae1d88a0a5b0752112c3488cc38431f50e8b937eef906f9ed
redyms	b6d19c3e6e82bbde62984f50144ce4d98a18871374ec5d313489d5831317c480
remtasu	7f9ae2fb8e0a2ef18ab16afcb5053e425a42c0caa8080210707893dfdf328936
rokku	438888ef36bad1079af79daf152db443b4472c5715a7b3da0ba24cc757c53499
rombrast	1e87d2cbc136d9695b59e67f37035a45a9ad30f5fccc216387a03c0a62afa9d4
rovnix	5180155031a3939607452619ba53f76bf2e1cecb9abd0b745192af378fa5b7cc
sakula	22f5fa60c2286e22bee79bcde6e9c7ee80b42ef308c6bb7aed6d6163e5da0214
sality2016	1d441a7c0f32513163b7e287cfa4869d61b750acf212449bc98193a30ea47d23
schwarzesonne	aa37ece103da4a5c94079b66d13d2112c908094fa3dfd70b89dc07b198f92985
sefnit	3b2e246d5460d214ad07d2b395b95b77e0679f46dc563e6eb2ff067c650fad8c
shifu	01c53e0d31c578393ba09add090fff2560c1f53a2a13fdbed8a66bd783a2ee70
shylock	35ccf0e051fb0ff61e52ff4130eb38521f954f90ad9c97be59af1e7901974557
sinowal	c7a65cea20c1a312c3619ed5923c2d1eee55afc26f21f1ce9e01179aec0db3ad
skynet	880ce96c128e416c30e9786f5b453030137ee2ef9ee9abf32267d3d6444c4119
sopinar	f711606b253aa4faf3c1e0f9b8f6bbb9e246ce1a0320ab779833bbc166d63ac1
soraya	c1a2a7bbd65d61a6ecd77fe9e921e39fad770a8cbcf15ea7ba0cb9066ce1656c
sphinx	792c968786d7b161232a683f1b38beb83e26807d4e9741638291c46478dac206
symmi	8f98246c03a8e6bf2c342cb5ebc9e82e0cb596c333e5cab0eaed2dc247378755
teerac	447fe9c756081e328f5855b60f3f5c1af460c6d86df0bc671556c60515531fa1
tempedreve	77f24ca7bbbfdcd4d182d794e19243ada9e97e60e36ad091797c500e5c4de6af
tinba	09478bf4833505d3d7b66d4f30ccce6b9fde3ea51b9ccf6fdeadc008efba43d8
tinybanker	59ac093c3a08619fa16d1d8a06c23f7aa63ec01da1e1ecf979421a355d595e8e
trustezeb	1f8992bb765566cce72dfc77c9f54916480d87aa70b88994333578d23c06e1dc
tuscas	9ef9d58f869f6b737383fb3b659e48910a7593d720e9f6479d618d396b5ddc56
urausy	e4f821e0c469ef6030f1b309a56a313f25a88e4f91a3ef28387785bb07ebcd8e
urlzone	a639e0d31e44fbb6786733325da5938b8086877f893b6d1cdb78d40e50d32a4b
ursnif	30a13051c39c77bf1a5fa0bcd170185cfc763fc82a5a9a6153e3b3b9198c199b
vawtrack2016	b6260c3f504ebab55a1c13bfdc88c04ea20265a661f3445a6c040c91556cbf4f
vawtrak	3547187819ee0ab9c72530dc5dbe79b81c84dbc4a57a274e0044ce997fb6982b
vundo	3563ce2be8aa67950875c8b0678ed79636bbe7e26215f8a25f32e316a70a6e25
xswkit	34f9e77c56549fa25b02ae97891054b963b4bc9e32971b65cc6f203e8a2792f1
zeroaccess	be00ef4081af7d101b36e4552e6ca7ba2cd91f906cc5dff8751ef2078026a774
zeusp2p	4bbf0a795cf4d66af449723332c94c9c4ba0edd7ae23809259bab82375b7da81
zurgop	d12526fc430fa213d77f8523a89c92c5f4e0d11deacbaf5c160a16f87ed5adc3

TABLE C.3: Malware families of Quincy’s Windows 10 evaluation

Feature	Windows XP	Windows 7	Windows 10
$counter_{debugger}$	✓	✗	✗
$counter_{sandbox}$	✓	✗	✓
$counter_{vm}$	✗	✗	✗
$api_{general_api_strings}$	✓	✓	✓
api_{hbcias}	✓	✓	✓
$api_{dynamic_loading}$	✓	✓	✗
$api_{hashing}$	✓	✓	✓
$binary_{has_header}$	✓	✓	✗
$binary_{wiped_header}$	✗	✗	✗
$binary_{imports}$	✗	✗	✗
$binary_{is_pe_or_dll}$	✓	✓	✓
$binary_{is_module}$	✓	✓	✗
$binary_{exports}$	✗	✗	✗
$binary_{is_dynamic_library}$	✗	✗	✓
$code_{functions}$	✓	✓	✗
$code_{shellcode}$	✓	✓	✓
$code_{indirect_calls}$	✓	✓	✓
$code_{indirect_jumps}$	✓	✓	✓
$code_{hooks}$	✓	✓	✓
$crypto_{cipher}$	✗	✗	✗
$crypto_{encoding}$	✓	✗	✗
$crypto_{hashing}$	✗	✓	✗
$memory_{is_sparse}$	✓	✓	✓
$memory_{high_entropy_areas}$	✓	✓	✓
$memory_{is_heap}$	✓	✗	✗
$memory_{protection}$	✓	✓	✓
$memory_{mapped}$	✓	✗	✗
$memory_{tag}$	✓	✓	✗
$memory_{threads}$	✓	✓	✓
$memory_{private}$	✓	✓	✓
$memory_{embedded_executable}$	✗	✗	✗
$memory_{english_strings}$	✓	✗	✗
$memory_{network_strings}$	✓	✓	✓
$memory_{victim_strings}$	✓	✗	✗
$memory_{persistence}$	✓	✗	✓
$trojan_{banking}$	✗	✗	✗
$trojan_{cookies}$	✗	✗	✗
$trojan_{credentials}$	✓	✓	✗
Total	28	21	17

TABLE C.4: Features that were selected for the final models of *Quincy* on Windows XP, Windows 7, and Windows 10

Abbreviations

APC	**A**synchronous **P**rocedure **C**all
APT	**A**dvanced **P**ersistent **T**hreat
API	**A**pplication **P**rogramming **I**nterface
ASLP	**A**dress **S**pace **L**ayout **P**ermutation
ASLR	**A**ddress **S**pace **L**ayout **R**andomization
AUC	**A**rea **U**nder **C**urve
CART	**C**lassification **A**nd **R**egression **T**rees
CFI	**C**ontrol **F**low **I**ntegrity
CPU	**C**entral **P**rocessing **U**nit
DDoS	**D**istributed **D**enial **o**f **S**ervice
DEP	**D**ata **E**xecution **P**revention
DLL	**D**ynamic-**L**ink **L**ibrary
DoS	**D**enial **o**f **S**ervice
FP	**F**alse **P**ositive
FPR	**F**alse **P**ositive **R**ate
FN	**F**alse **N**egative
HBCIA	**H**ost-**B**ased **C**ode **I**njection **A**ttack
I/O	**I**nput/**O**utput
IoC	**I**ndicator **o**f **C**ompromise
IPS	**I**ntrusion **P**revention **S**ystem
ISR	**I**struction **S**et **R**andomization
MSE	**M**icrosoft **S**ecurity **E**ssentials
OS	**O**perating **S**ystem
PCA	**P**rincipal **C**omponent **A**nalysis
PE	**P**ortable **E**xecutable

PEB **P**rocess **E**nvironment **B**lock

PoC **P**roof **o**f **C**oncept

PoS **P**oint **o**f **S**ale

PRNG **P**seudo **R**andom **N**umber **G**enerator

RAT **R**emote **A**dministration **T**ool

RCIA **R**emote **C**ode **I**njection **A**ttack

RFE **R**ecursive **F**eature **E**limination

ROC **R**eceiver **O**perating **C**haracteristic

ROP **R**eturn **O**riented **P**rogramming

RWX **R**ead **W**rite **E**xecute

SICE **S**hotgun **I**njection **C**oncurrent **E**xecution

SITM **S**hotgun **I**njection **T**hread **M**anipulation

SVM **S**upport **V**ector **M**achine

TICE **T**argeted **I**njection **C**oncurrent **E**xecution

TITM **T**argeted **I**njection **T**hread **M**anipulation

TP **T**rue **P**ositive

TPR **T**rue **P**ositive **R**ate

TN **T**rue **N**egative

VM **V**irtual **M**achine

VMI **V**irtual **M**achine **I**ntrospection

List of Figures

2.1 Decision tree for choosing the right football club. Brown nodes represent tests and green leafs represent decisions. 17
2.2 Predicting a class using an ensemble classifier E with five weak classifiers E_i, where $i \in \{1, .., 5\}$. 18

4.1 Simplified HBCIA algorithm comprising three steps. First, the algorithm selects a victim. Second, it copies the code into the victim. Third, it triggers the code execution within the victim. Possible failure states are omitted. 46
4.2 Concurrent Execution: A dropper injects (ϵ_{inject}) code into a victim process that runs a benign program (ϵ_{victim}). The dropper and the victim comprise at first one thread (self-pointing arrow). After successful injection the dropper terminates and the victim runs a malicious thread concurrently to its original thread. This thread originated in the injected payload. 51
4.3 Thread Manipulation: A dropper (ϵ_{inject}) hollows a program (ϵ_{victim}). The dropper and the victim comprise at first one thread (self-pointing arrow). After successful injection the dropper terminates and the victim continues with one thread. However, now it executes the payload instead of the original program. 52

5.1 Daily number of samples from 2013-03-21 until 2014-06-19 of the *VirusSign* data set. *VirusSign* provided 500 samples daily. However, there were periods where no samples were available, e.g. in March 2014. 61
5.2 Top five families over time from 2013-03-21 until 2014-06-19 of the *VirusSign* data set. This data set reflects, for instance, the takedown of the *Sirefef* botnet in December 2013. 62
5.3 Top 25 families from 2013-03-21 until 2014-06-19 of our data set. 63
5.4 Weekly percentage of HBCIA-employing malware from 2013-03-21 until 2014-06-19 as a scatter plot. 64
5.5 Relative distribution of victim processes on Windows XP. Most HBCIAs targeted *explorer.exe*, which is a Windows system process that always runs. 67
5.6 The temporal distribution of the eight HBCIA-employing malware families of our data set ranging from 2007 to 2013. 70

6.1 Overview of Bee Master's architecture: the *Queen Bee* and its *Worker Bees*. 81
6.2 Control flow of the *Queen Bee's Worker Bee* handling algorithm. 82
6.3 Methodology of the evaluation of *Bee Master* 91

7.1 *Quincy* receives memory dumps with labeled memory regions as input. ① Then, it extracts 36 HBCIA-related features from seven categories. ② Subsequently, it discards invaluable features and embeds the rest in a vector space. ③ Next, it induces a binary classifier. ④ Finally, it classifies unseen memory areas. 104
7.2 Isomaps of Windows XP (left) and Windows 7 (right). Green dots denote benign samples, red crosses denote malicious samples. The samples scatter along two main axes. 120
7.3 Isomap of Windows 10. Green dots denote benign samples, red crosses denote malicious samples. The samples are broader scattered than on Windows XP and Windows 7. 121
7.4 Flow chart of one fold of the 10-fold cross validation loop. 123
7.5 Top 15 features on Windows XP (top left), Windows 7 (top right), and Windows 10 (bottom), ranked by their relative feature importance. We estimated the values during the recursive feature elimination with *Random Forests*. 126
7.6 ROC curves of top five classifiers on Windows XP (left) and Windows 7 (right). *Extremely Randomized Trees* and *Random Forest* rank first on both Windows versions. 129
7.7 ROC curves of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on Windows XP (left) and Windows 7 (right) 130
7.8 ROC curves of top five classifiers on Windows 10 (left) and *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on Windows 10 (right) 131
7.9 Averaged performance of *Quincy* (green), *Malfind* (orange), and *Hollowfind* (red) in temporal evaluation on Windows XP, Windows 7, and Windows 10 as ROC curve. 133

List of Tables

4.1 Taxonomy of HBCIA algorithms: Three of the four classes can be found in real world malware. Note that there may be **SITM**-employing malware in the future that is fully ROP-based (see Discussion in Section 4.4.) 54

5.1 Listing of considered publicly available malware research data sets 58
5.2 Matching of publicly available data sets to our four requirements **R1**-**R4** of Section 5.1. The symbol ✓denotes that a data set fulfills a certain requirement, the symbol ✗denotes that a data set does not fulfill a certain requirement. 59
5.3 Matching of publicly available data sets to our two requirements of Section 5.3 to a corpus for corroborating the family feature hypothesis. 69
5.4 Summary of the data set for the family feature investigation of Section 5.3. 72
5.5 Matching of publicly available data sets to our two requirements **R1** and **R2** of Section 5.4. 74
5.6 The distribution of malware families to HBCIA algorithm classes in absolute figures and percentage. All classes are populated except the **SITM** class. 75

6.1 Matching of most related approaches with our five requirements to dynamically detect HBCIAs. Each requirement can be either fulfilled (●), partially fulfilled (◐) or unfulfilled (○). 80
6.2 Comparison of the three different privilege levels of the implementation. Each category has three relative degrees: low (○), medium (◐), and high (●). Furthermore, the table lists whether (✓) or not (✗) an implementation allows bare-metal deployment. 85
6.3 Matching of publicly available data sets to our requirements of Section 6.2.3. 90
6.4 The number of working malware families and the families that refuse to work on the Windows versions. 92
6.5 Summary of the Windows goodware per Windows version. The data set comprises Windows system tools and popular portable apps. 93
6.6 *1001-injects*: summary of injection techniques for Linux. The proof of concept implementation is hosted on github [265]. 94
6.7 Summary of the observed injections into the *Worker Bees* employed on Windows XP, 7, and 8. 95
6.8 Distribution of families to the amount of attacked processes on Windows XP. 96

7.1 Matching of related approaches to our four requirements to statically detect HBCIAs. Each requirement can be either fulfilled (●), partially fulfilled (◐), or unfulfilled (○). 102
7.2 Overview of *Quincy*'s 36 features in categorical and alphabetical order. The features are ranked based on the *Recursive Feature Selection* (see Section 7.2.3). Quincy's final models may not utilize all features (see Appendix C). 105
7.3 Matching of supervised machine learning algorithm classes to our requirements. We listed at least one representative per class. Three possible results: does not match (○), partially matches (◐), and matches (●). . . . 114
7.4 Comparing open source malware data sets with our four requirements to an evaluation data set. 117
7.5 Summary of the three data sets of Windows XP, 7, and 10. Benign memory regions may be duplicates, e.g. system libraries. The amount of memory regions constantly increases from version to version. 119
7.6 Summary of the algorithms' hyperparameters that we chose to optimize. The total number of features is denoted by f. ∞ denotes that there was no depth limit, the algorithm determined the optimal depth. 124
7.7 Summary of the final machine learning algorithm ranking on Windows XP, 7 and 10. The ranking is based on the final evaluation on $d_{validation}$. . 127
7.8 Final data of the evaluation of *Quincy* with *AdaBoost*, *Decision Tree*, *Extremely Randomized Trees*, and *Gradient Boosting* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). 128
7.9 Final data of the evaluation of *Quincy* with *KNN*, *Random Forest*, and *SVM* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). 128
7.10 Final data of the evaluation of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). 128
7.11 Final data of the temporal evaluation of *Quincy* with *Extremely Randomized Trees* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). 128
7.12 Family detection and family completeness of *Quincy* with *Extremely Randomized Trees*, *Malfind*, and *Hollowfind* on $d_{validation}$. 132
7.13 Example case of family detection and completeness: Split 2 of the Windows 7 evaluation. The table lists the results of *Quincy*, *Malfind*, and *Hollowfind*. Fully detected families are marked in green, partially detected in yellow, and undetected families in red. 132
7.14 Parameters of final *Extremely Randomized Trees* models. We created these models on the whole data set of each operating system. During a randomized grid search, two parameters were optimized (number of trees in ensemble and maximal features). 134

A.1 Malware families considered in HBCIA algorithm prevalence estimation . 146

B.1 Representatives of malware families 148

C.1 Malware families of Quincy's Windows XP evaluation 150

C.2 Malware families of Quincy's Windows 7 evaluation 151
C.3 Malware families of Quincy's Windows 10 evaluation 152
C.4 Features that were selected for the final models of *Quincy* on Windows XP, Windows 7, and Windows 10 153

Bibliography

[1] Cisco. Seize New IoT Opportunities with the Cisco IoT System. http://www.cisco.com/web/solutions/trends/iot/portfolio.html, 2015. [Online; accessed February 2018].

[2] Darren Murph. Google clarifies 18 month Android upgrade program, details far from solidified. http://www.engadget.com/2011/05/10/google-clarifies-18-month-android-upgrade-program-details-far-f/, 2011. [Online; accessed February 2018].

[3] PandaLabs. PandaLabs neutralized 75 million new Malware Samples in 2014, twice as many as in 2013. http://www.pandasecurity.com/mediacenter/press-releases/ pandalabs-neutralized-75-million-new-malware-samples-2014-twice-many-2013/, 2015. [Online; accessed February 2018].

[4] PandaLabs. 27 percent of all recorded malware appeared in 2015. http://www.pandasecurity.com/mediacenter/press-releases/all-recorded-malware-appeared-in-2015/, 2016. [Online; accessed February 2018].

[5] Intel Security. Net Losses: Estimating the Global Cost of Cybercrime. http://www.mcafee.com/de/resources/reports/rp-economic-impact-cybercrime2.pdf, 2014. [Online; accessed February 2018].

[6] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. Duqu: Analysis, detection, and lessons learned. In *Proceedings of European Workshop on System Security (EuroSec)*, 2012.

[7] Aleksandr Matrosov and Eugene Rodionov. Stuxnet under the microscope. https://go.eset.com/us/resources/white-papers/Stuxnet_Under_the_Microscope.pdf, 2010. [Online; accessed February 2018].

[8] NetMarketShare. Desktop Operating System Market Share. https://www.netmarketshare.com/operating-system-market-share.aspx, 2017. [Online; accessed February 2018].

[9] Veo Zhang. Hacking Team RCS Android Spying Tool Listens to Calls; Roots Devices to Get In. http://blog.trendmicro.com/trendlabs-security-intelligence/hacking-team-rcsandroid-spying-tool-listens-to-calls-roots-devices-to-get-in/, 2015. [Online; accessed February 2018].

[10] Limor Kessem. Thieves Reaching for Linux: Hand of Thief Trojan Targets Linux. https://blogs.rsa.com/thieves-reaching-for-linux-hand-of-thief-trojan-targets-linux-inth3wild, 2013. [Online; accessed February 2018].

[11] Marc-Etienne M.Léveillé. OSX/Flashback - The first malware to infect hundreds of thousands of Apple Mac. http://go.eset.com/us/resources/white-papers/osx_flashback.pdf, 2012. [Online; accessed February 2018].

[12] Geoff McDonald, Liam O Murchu, Stephen Doherty, and Eric Chien. Stuxnet 0.5: The missing link. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/stuxnet_0_5_the_missing_link.pdf, 2013. [Online; accessed February 2018].

[13] Kaspersky Lab. Unveiling "Careto" - The Masked APT. http://kasperskycontenthub.com/wp-content/uploads/sites/43/vlpdfs/unveilingthemask_v1.0.pdf, 2014. [Online; accessed February 2018].

[14] ITU ICT. ITU ICT Facts and Figures – The world in 2015 . http://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2015.pdf, 2016. [Online; accessed February 2018].

[15] Thomas Barabosch and Elmar Gerhards-Padilla. Host-based Code Injection Attacks: A popular technique used by malware. In *Proceedings of the 9th International Conference on Malicious and Unwanted Software (MALWARE)*, 2014. URL https://doi.org/10.1109/MALWARE.2014.6999410.

[16] Thomas Barabosch, Sebastian Eschweiler, and Elmar Gehards-Padilla. Bee Master: Detecting Host-Based Code Injection Attacks. In *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2014. URL https://doi.org/10.1007/978-3-319-08509-8_13.

[17] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy: Detecting Host-Based Code Injection Attacks in Memory Dumps. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017. URL https://doi.org/10.1007/978-3-319-60876-1_10.

[18] F. Baiardi and D. Sgandurra. An Obfuscation-Based Approach against Injection Attacks. In *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*, 2011. URL https://doi.org/10.1109/ARES.2011.17.

[19] Kevin Z Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *Proceedings of the 20th USENIX Security Symposium*, 2011. URL http://dl.acm.org/citation.cfm?id=2028067.2028076.

[20] Hung-Min Sun, Yu-Tung Tseng, and Yue-Hsun Lin. Detecting the Code Injection by Hooking System Calls in Windows Kernel Mode. In *Proceedings of the International Computer Symposium (ICS)*, 2006. URL https://www.researchgate.net/publication/267828178_Detecting_the_Code_Injection_by_Hooking_System_Calls_in_Windows_Kernel_Mode.

[21] Abhinav Srivastava and Jonathon Giffin. Automatic discovery of parasitic malware. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2010. URL https://doi.org/10.1007/978-3-642-15512-3_6.

[22] Armin Buescher, Felix Leder, and Thomas Siebert. Banksafe: Information Stealer Detection Inside the Web Browser. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2011. URL http://dx.doi.org/10.1007/978-3-642-23644-0_14.

[23] A. Papadogiannakis, L. Loutsis, V. Papaefstathiou, and S. Ioannidis. ASIST: Architectural Support for Instruction Set Randomization. *Proceedings of the 20th Conference on Computer and Communications Security (CCS)*, 2013. URL http://doi.acm.org/10.1145/2508859.2516670.

[24] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th Conference on Computer and Communications Security (CCS)*, 2003. URL http://doi.acm.org/10.1145/948109.948146.

[25] Volatile Systems. The Volatility Framework: Volatile memory artifact extraction utility framework. https://www.volatilesystems.com/default/volatility. [Online; accessed February 2018].

[26] Andrew White, Bradley Schatz, and Ernest Foo. Integrity verification of user space code. In *Proceedings of the 13th Digital Forensics Research Conference (DFRWS)*, 2013. URL https://doi.org/10.1016/j.diin.2013.06.007.

[27] K A Monnappa. Detecting Deceptive Process Hollowing Techniques Usind Hollowfind Volatility Plugin. https://cysinfo.com/detecting-deceptive-hollowing-techniques/, 2016. [Online; accessed February 2018].

[28] Gábor Pék, Zsombor Lázár, Zoltán Várnagy, Márk Félegyházi, and Levente Buttyán. Membrane: A Posteriori Detection of Malicious Code Loading by Memory Paging Analysis. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, 2016. URL http://doi.org/10.1007/978-3-319-45744-4_10.

[29] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy implementation on github. https://github.com/tbarabosch/quincy, 2017. [Online; accessed February 2018].

[30] Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla. Quincy complementary material on github. https://github.com/tbarabosch/quincy-complementary-material, 2017. [Online; accessed February 2018].

[31] Merriam-Webster. Merriam-Webster Online Dictionary. http://www.merriam-webster.com. [Online; accessed February 2018].

[32] Simon Kramer and Julian C Bradfield. A general definition of malware. *Journal in computer virology*, 6(2):105–114, 2010. URL https://doi.org/10.1007/s11416-009-0137-1.

[33] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, and Lingyu Wang. On the analysis of the zeus botnet crimeware toolkit. In *Proceedings of the 8th Annual International Conference on Privacy Security and Trust (PST)*, 2010. URL https://doi.org/10.1109/PST.2010.5593240.

[34] Felix C. Freiling, Thorsten Holz, and Georg Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of 10th European Symposium on Research in Computer Security (ESORICS)*, 2005. URL http://dx.doi.org/10.1007/11555827_19.

[35] Daniel Plohmann and Elmar Gerhards-Padilla. Case study of the Miner botnet. In *Proceedings of the 4th International Conference on Cyber Conflict (CyCon)*, 2012. URL http://ieeexplore.ieee.org/document/6243985/.

[36] Tinba source code. https://github.com/nyx0/Tinba, 2014. [Online; accessed February 2018].

[37] Rovnix source code. https://github.com/nyx0/Rovnix, 2014. [Online; accessed February 2018].

[38] Zeus source code. https://github.com/Visgean/Zeus, 2011. [Online; accessed February 2018].

[39] Juan Caballero and Chris Grier and Christian Kreibich and Vern Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Security Symposium*, 2011. URL https://doi.org/10.1007/s11416-009-0137-1.

[40] Tyler Moore, Richard Clayton, and Ross Anderson. The economics of online crime. *The Journal of Economic Perspectives*, 23(3):3–20, 2009. URL https://doi.org/10.1257/jep.23.3.3.

[41] Symantec. Regin: Top-tier espionage tool enables stealthy surveillance. http://www.symantec.com/content/en/us/enterprise/media/security_

response/whitepapers/regin-analysis.pdf, 2015. [Online; accessed February 2018].

[42] Morgan Marquis-Boire, Claudio Guarnieri, and Ryan Gallagher. Secret Malware in European Union Attack Linked to U.S. and British Intelligence. https://theintercept.com/2014/11/24/secret-regin-malware-belgacom-nsa-gchq/, 2014. [Online; accessed February 2018].

[43] Syrian Malware Samples - Binaries from the Syrian revolution. http://syrianmalware.com, 2014. [Online; accessed February 2018].

[44] The Citizen Lab. Malware Attack Targeting Syrian ISIS Critics. https://citizenlab.org/2014/12/malware-attack-targeting-syrian-isis-critics/, 2014. [Online; accessed February 2018].

[45] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540654100.

[46] Felix Leder and Tillmann Werner. Know Your Enemy: Containing Conficker - To Tame A Malware. https://www.honeynet.org/files/KYE-Conficker.pdf, 2009. [Online; accessed February 2018].

[47] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the 17th Symposium on Network and Distributed System Security (NDSS)*, 2010. URL http://www.eurecom.fr/publication/3022.

[48] Tal Garfinkel and Keith Adams and Andrew Warfield and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2007. URL http://dl.acm.org/citation.cfm?id=1361397.1361403.

[49] Peter Ferrie. The ultimate anti-debugging reference. http://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf, 2011. [Online; accessed February 2018].

[50] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 1st edition, 2012. ISBN 9781593272906.

[51] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Symposium on Network and Distributed System Security (NDSS)*, 2003. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.11.8367.

[52] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *Proceedings of the 28th Symposium on Security and Privacy (S&P)*, 2007. URL http://dx.doi.org/10.1109/MSP.2007.45.

[53] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. BareBox: Efficient Malware Analysis on Bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011. URL http://doi.acm.org/10.1145/2076732.2076790.

[54] Tarik Soulami. *Inside Windows Debugging: Practical Debugging and Tracing Strategies*. Microsoft Press, 2012. ISBN 9780735662780.

[55] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th Symposium on Security and Privacy (S&P)*, 2015. URL https://doi.org/10.1109/SP.2015.46.

[56] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary Obfuscation Using Signals. In *Proceedings of the 16th USENIX Security Symposium*, 2007. URL http://dl.acm.org/citation.cfm?id=1362903.1362922.

[57] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the 15th Symposium on Network and Distributed System Security (NDSS)*, 2008. URL citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.2395.

[58] Prabhanjan Ananth, Divya Gupta, Yuval Ishai, and Amit Sahai. Optimizing Obfuscation: Avoiding Barrington's Theorem. In *Proceedings of the 21th Conference on Computer and Communications Security (CCS)*, 2014. URL http://doi.acm.org/10.1145/2660267.2660342.

[59] Paul Baecher, Markus Koetter, Maximillian Dornseif, and Felix Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006. URL https://doi.org/10.1007/11856214_9.

[60] Jesse Kornblum. Identifying Almost Identical Files Using Context Triggered Piecewise Hashing. *Digital investigation*, 3:91–97, 2006. URL http://dx.doi.org/10.1016/j.diin.2006.06.015.

[61] Georg Wicherski. peHash: A Novel Approach to Fast Malware Clustering. In *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More (LEET)*, 2009. URL http://dl.acm.org/citation.cfm?id=1855676.1855677.

[62] FireEye. Tracking Malware with Import Hashing. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html, 2014. [Online; accessed February 2018].

[63] Robert Lyda and James Hamrock. Using entropy analysis to find encrypted and packed malware. *Security and Privacy*, 5(2):40–45, 2007. URL https://doi.org/10.1109/MSP.2007.48.

[64] Kevin Coogan, Saumya Debray, Tasneem Kaochar, and Gregg Townsend. Automatic Static Unpacking of Malware Binaries. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, 2009. URL http://dx.doi.org/10.1109/WCRE.2009.24.

[65] Hex-Rays Decompiler. https://www.hex-rays.com/products/decompiler/index.shtml. [Online; accessed February 2018].

[66] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 2014.

[67] Google. Rekall. http://www.rekall-forensic.com. [Online; accessed February 2018].

[68] Brijnesh Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the 35th*

Symposium on Security and Privacy (S&P), 2014. URL http://dx.doi.org/10.1109/SP.2014.45.

[69] Peter M. Chen and Brian D. Noble. When Virtual Is Better Than Real. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS)*, 2001. URL http://dl.acm.org/citation.cfm?id=874075.876409.

[70] Yangchun Fu and Zhiqiang Lin. Space Traveling across VM: Automatically Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection. In *Proceedings of the 33th Symposium on Security and Privacy (S&P)*, 2012. URL https://doi.org/10.1109/SP.2012.40.

[71] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32th Symposium on Security and Privacy (S&P)*, 2011. URL http://dx.doi.org/10.1109/SP.2011.11.

[72] Oracle VM VirtualBox. https://www.virtualbox.org, . [Online; accessed February 2018].

[73] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0321108957.

[74] Ari Juels and Ronald L. Rivest. Honeywords: Making Password-cracking Detectable. In *Proceedings of the 21th Conference on Computer and Communications Security (CCS)*, 2013. URL http://doi.acm.org/10.1145/2508859.2516671.

[75] D. Ramsbrock, R. Berthier, and Michel Cukier. Profiling Attacker Behavior Following SSH Compromises. In *37th Annual /IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007. URL https://doi.org/10.1109/DSN.2007.76.

[76] Sebastian Poeplau. Ghost USB honeypot. https://github.com/honeynet/ghost-usb-honeypot, 2012. [Online; accessed February 2018].

[77] Angelo Dell'Aera. Thug. https://github.com/buffer/thug, 2015. [Online; accessed February 2018].

[78] Niels Provos and Thorsten Holz. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*. Addison-Wesley Professional, first edition, 2007. ISBN 9780321336323.

[79] dionaea - catches bugs. http://dionaea.carnivore.it, 2009. [Online; accessed February 2018].

[80] The Honeynet Project. Capture-HPC Client Honeypot. https://projects.honeynet.org/capture-hpc/, 2008. [Online; accessed February 2018].

[81] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., 2 edition, 2003. ISBN 9780136042594.

[82] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012. ISBN 9781107422223.

[83] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine learning*, 20(3), 1995. URL http://dx.doi.org/10.1023/A:1022627411411.

[84] Thomas Cover and Peter Hart. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967.

[85] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 2006. ISBN 0387310738.

[86] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., 2001. ISBN 9780387848570.

[87] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984. ISBN 9780412048418.

[88] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2005. ISBN 1558609016.

[89] Leo Breiman. Random Forests. In *Journal of Machine learning*, volume 45. Springer, 2001. URL http://dx.doi.org/10.1023/A:1010933404324.

[90] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely Randomized Trees. *Journal of Machine learning*, 63(1):3–42, 2006. URL http://dx.doi.org/10.1007/s10994-006-6226-1.

[91] Anh Nguyen-Tuong, Salvatore Guarnieri, Doug Greene, Jeff Shirley, and David Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Security and Privacy in the Age of Ubiquitous Computing*, volume 181 of *IFIP Advances in Information and Communication Technology*, pages 295–307. Springer, 2005.

[92] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the 14th Symposium on Network and Distributed System Security (NDSS)*, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.72.4505.

[93] Yves Younan, Wouter Joosen, and Frank Piessens. A Methodology for Designing Countermeasures against Current and Future Code Injection Attacks. In *International Workshop on Information Assurance*, 2005.

[94] Justin Ma, John Dunagan, Helen J Wang, Stefan Savage, and Geoffrey M Voelker. Finding Diversity in Remote Code Injection Exploits. In *Proceedings of the 6th Internet Measurement Conference (IMC)*, 2006. URL http://doi.acm.org/10.1145/1177080.1177087.

[95] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. An empirical study of real-world polymorphic code injection attacks. In *Proceedings of the 2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, 2009.

[96] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX)*, 2000. URL https://doi.org/10.1109/DISCEX.2000.821514.

[97] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998. URL http://dl.acm.org/citation.cfm?id=1267549.1267554.

[98] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. Transparent run-time defense against stack-smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, 2000. URL http://dl.acm.org/citation.cfm?id=1267724.1267745.

[99] David Wagner, Jeffrey S Foster, Eric A Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 7th Symposium on Network and Distributed System Security (NDSS)*, 2000.

[100] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection From printf Format String Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001. URL http://dl.acm.org/citation.cfm?id=1267612.1267627.

[101] Wei Hu, Jason Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-tuong, and Jonathan C. Rowanhill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, 2006. URL http://doi.acm.org/10.1145/1134760.1134764.

[102] Jonathan Pincus and Brandon Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *Proceedings of the 25th Symposium on Security and Privacy (S&P)*, 2004. URL http://dx.doi.org/10.1109/MSP.2004.36.

[103] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant Program Execution: Using Multi-core Systems to Defuse Buffer-Overflow Vulnerabilities. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, 2008. URL http://dx.doi.org/10.1109/CISIS.2008.136.

[104] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, 2009. URL http://doi.acm.org/10.1145/1519065.1519071.

[105] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, 2003. URL http://dl.acm.org/citation.cfm?id=1251353.1251371.

[106] Cullen Linn, Mohan Rajagopalan, Scott Baker, Christian S Collberg, Saumya K Debray, and John H Hartman. Protecting Against Unexpected System Calls. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251414.

[107] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th Conference on Computer and Communications Security (CCS)*, 2007. URL http://doi.acm.org/10.1145/1315245.1315313.

[108] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th Conference on Computer and Communications Security (CCS)*, 2008. URL http://doi.acm.org/10.1145/1455770.1455776.

[109] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically Returning to Randomized Lib(C). In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, 2009. URL http://dx.doi.org/10.1109/ACSAC.2009.16.

[110] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011. URL http://dx.doi.org/10.1007/978-3-642-23644-0_7.

[111] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010. URL http://doi.acm.org/10.1145/1920261.1920269.

[112] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2011. URL http://doi.acm.org/10.1145/1966913.1966920.

[113] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with ”return-less” kernels. In *Proceedings of the*

5th European Conference on Computer Systems (EuroSys)*, 2010. URL http://doi.acm.org/10.1145/1755913.1755934.

[114] V. Pappas, M. Polychronakis, and A.D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 33th Symposium on Security and Privacy (S&P)*, 2012. URL http://dx.doi.org/10.1109/SP.2012.41.

[115] Lixin Li, James E Just, and R Sekar. Address-Space Randomization for Windows Systems. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006. URL http://dx.doi.org/10.1109/ACSAC.2006.10.

[116] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th Conference on Computer and Communications Security (CCS)*, 2003. URL http://doi.acm.org/10.1145/948109.948147.

[117] The PAX Team. Pax project. https://pax.grsecurity.net. [Online; accessed February 2018].

[118] A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. https://support.microsoft.com/en-us/kb/875352, 2013. [Online; accessed February 2018].

[119] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems (SRDS)*, 2003. URL https://doi.org/10.1109/RELDIS.2003.1238076.

[120] Sandeep Bhatkar, Daniel DuVarney, and Ravi Sekar. Address Obfuscation: An Efficient Approach to Combat a Board Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, 2003. URL http://dl.acm.org/citation.cfm?id=1251353.1251361.

[121] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251415.

[122] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, 2006. URL http://dx.doi.org/10.1109/ACSAC.2006.9.

[123] Sandeep Bhatkar and R Sekar. Data space randomization. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008. URL http://dx.doi.org/10.1007/978-3-540-70542-0_1.

[124] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium*, 2012. URL http://dl.acm.org/citation.cfm?id=2362793.2362833.

[125] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th Symposium on Security and Privacy (S&P)*, 2013. URL http://dx.doi.org/10.1109/SP.2013.45.

[126] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proceedings of the 22nd Symposium on Network and Distributed System Security (NDSS)*, 2015.

[127] Eitaro Shioji, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, 2012. URL http://doi.acm.org/10.1145/2420950.2420996.

[128] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th Conference on Computer and Communications Security (CCS)*, 2004. URL http://doi.acm.org/10.1145/1030083.1030124.

[129] Tyler Durden. Bypassing PAX ASLR protection. *Phrack Magazine*, 59:9–9, 2002. URL http://phrack.org/issues/59/9.html. [Online; accessed February 2018].

[130] Xi Chen, Herbert Bos, and Cristiano Giuffrida. Codearmor: Virtualizing the code space to counter disclosure attacks. In *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*, 2017. URL https://doi.org/10.1109/EuroSP.2017.17.

[131] Frederick B Cohen. Operating System Protection Through Program Evolution. *Computers & Security*, 12(6):565–584, 1993. URL http://dx.doi.org/10.1016/0167-4048(93)90054-9.

[132] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *Transactions on Information and System Security*, 8:3–40, 2005. URL http://doi.acm.org/10.1145/1053283.1053286.

[133] Ana Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? The Effectiveness of Instruction Set Randomization. In *Proceedings of the 14th USENIX Security Symposium*, 2005. URL http://dl.acm.org/citation.cfm?id=1251398.1251408.

[134] Georgios Portokalidis and Angelos D Keromytis. Fast and Practical Instruction-set Randomization for Commodity Systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, 2010. URL http://doi.acm.org/10.1145/1920261.1920268.

[135] Stephen W Boyd, Gaurav S Kc, Michael E Locasto, Angelos D Keromytis, and Vassilis Prevelakis. On the General Applicability of Instruction-Set Randomization. *Transactions on Dependable and Secure Computing (TDSC)*, 7(3):255–270, 2010. URL https://doi.org/10.1109/TDSC.2008.58.

[136] Monica Chew and Dawn Song. Mitigating buffer overflows by operating system randomization. Technical report, Carnegie Mellon University (CMU), 2002.

[137] Yoshihiro Oyama and Akinori Yonezawa. Prevention of code-injection attacks by encrypting system call arguments, 2006. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.74.5899.

[138] Xuxian Jiang, Helen J. Wang, Dongyan Xu, and Yi min Wang. RandSys: Thwarting Code Injection Attacks with System Service Interface Randomization. In *Proceedings of the 26th International Symposium on Reliable Distributed Systems (SRDS)*, 2007. URL http://dl.acm.org/citation.cfm?id=1308172.1308236.

[139] Lynette Qu Nguyen, Tufan Demir, Jeff Rowe, Francis Hsu, and Karl N. Levitt. A framework for diversifying windows native APIs to tolerate code injection attacks. In *Proceedings of the 14th Conference on Computer and Communications Security (CCS)*, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.5580.

[140] Calton Pu. *A specialization toolkit to increase the diversity of operating systems*. PhD thesis, Portland State University, 1996. URL http://archives.pdx.edu/ds/psu/10637.

[141] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS)*, 1997. URL http://dl.acm.org/citation.cfm?id=822075.822408.

[142] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated software diversity. In *Proceedings of the 35th Symposium on Security and Privacy (S&P)*, 2014. URL https://doi.org/10.1109/SP.2014.25.

[143] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, 2002. URL http://dl.acm.org/citation.cfm?id=647253.720293.

[144] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure Program Execution via Dynamic Information Flow Tracking. *SIGARCH Computer Architecture News*, 32(5):85–96, October 2004. URL http://doi.acm.org/10.1145/1037947.1024404.

[145] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th Conference on Computer and Communications Security (CCS)*, 2005. URL http://doi.acm.org/10.1145/1102120.1102165.

[146] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *Transactions on Information and System Security (TISSEC)*, 13(1):4:1–4:40, November 2009. URL https://doi.org/10.1145/1609956.1609960.

[147] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006. URL http://dl.acm.org/citation.cfm?id=1298455.1298470.

[148] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 29th Symposium on Security and Privacy (S&P)*, 2008. URL https://doi.org/10.1109/SP.2008.30.

[149] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011. URL http://doi.acm.org/10.1145/2076732.2076783.

[150] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dong Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Proceedings of the 34th Symposium on Security and Privacy (S&P)*, 2013. URL http://dx.doi.org/10.1109/SP.2013.44.

[151] Mathias Payer, Antonio Barresi, and ThomasR. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015. URL http://dx.doi.org/10.1007/978-3-319-20550-2_8.

[152] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS)*, 2015. URL http://doi.acm.org/10.1145/2810103.2813676.

[153] Vishwath Mohan, Per Larsen, Stefan Brunthaler, K Hamlen, and Michael Franz. Opaque control-flow integrity. In *Proceedings of the 22nd Symposium*

on Network and Distributed System Security (NDSS), 2015. URL http://www.internetsociety.org/doc/opaque-control-flow-integrity.

[154] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[155] Lucas Davi, Daniel Lehmann, Ahmad-Reza Sadeghi, and Fabian Monrose. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671251.

[156] Enes Goktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 35th Symposium on Security and Privacy (S&P)*, 2014. URL http://dx.doi.org/10.1109/SP.2014.43.

[157] Ben Niu and Gang Tan. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS)*, 2015. URL http://doi.acm.org/10.1145/2810103.2813644.

[158] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS)*, 2015. URL http://doi.acm.org/10.1145/2810103.2813646.

[159] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS)*, 2015. URL http://doi.acm.org/10.1145/2810103.2813671.

[160] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Security Symposium*, 2015. URL http://dl.acm.org/citation.cfm?id=2831143.2831154.

[161] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *Proceedings of the 24th Symposium on Security and Privacy (S&P)*, 2003. URL http://dl.acm.org/citation.cfm?id=829515.830554.

[162] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and Hartman P. Moseley. A MultiFaceted Defence Mechanism Against Code Injection Attacks. In *Proceedings of the 11th Conference on Computer and Communications Security (CCS)*, 2004.

[163] Paruj Ratanaworabhan, V. Benjamin Livshits, and Benjamin G. Zorn. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the 18th USENIX Security Symposium*, 2009. URL http://dl.acm.org/citation.cfm?id=1855768.1855779.

[164] Babak Salamat, Todd Jackson, Gregor Wagner, and Christian Wimmer. Runtime Defense against Code Injection Attacks Using Replicated Execution. *Transactions on Dependable and Secure Computing*, 8:588–601, 2011. URL http://dx.doi.org/10.1109/TDSC.2011.18.

[165] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code Pointer Masking: Hardening Applications Against Code Injection Attacks. In *Proceedings of the 8th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2011. URL http://dl.acm.org/citation.cfm?id=2026647.2026664.

[166] Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. CPM: Masking Code Pointers to Prevent Code Injection Attacks. *Transactions on Information and System Security (TISSEC)*, 16(1):1:1–1:27, 2013. URL http://doi.acm.org/10.1145/2487222.2487223.

[167] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. 2016. URL https://doi.org/10.1109/SP.2016.12.

[168] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th Security Symposium (USENIX Security*

16), 2016. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun.

[169] Alexander Hanel. injdmp. http://hooked-on-mnemonics.blogspot.jp/p/injdmp.html, 2013. [Online; accessed February 2018].

[170] Wei Xu, Sandeep Bhatkar, and R Sekar. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *Proceedings of the 15th USENIX Security Symposium*, 2006. URL http://dl.acm.org/citation.cfm?id=1267336.1267345.

[171] Raoul Strackx and Frank Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 19th Conference on Computer and Communications Security (CCS)*, 2012. URL http://doi.acm.org/10.1145/2382196.2382200.

[172] David Korczynski and Heng Yin. Capturing malware propagations with code injections and code-reuse a acks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[173] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the 17th Symposium on Security and Privacy (S&P)*, 1996. URL http://dl.acm.org/citation.cfm?id=525080.884258.

[174] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *Proceedings of the 20th Symposium on Security and Privacy (S&P)*, 1999. URL https://www.researchgate.net/publication/2448365_Detecting_Intrusions_Using_System_Calls_Alternative_Data_Models.

[175] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the 22nd Symposium on Security and Privacy (S&P)*, 2001. URL http://dl.acm.org/citation.cfm?id=882495.884433.

[176] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 22th Symposium on Security and Privacy (S&P)*, 2001. URL http://dl.acm.org/citation.cfm?id=882495.884434.

[177] Roland HC Yap. Improving host-based IDS with argument abstraction to prevent mimicry attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006. URL http://dx.doi.org/10.1007/11663812_8.

[178] Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2007. URL http://dl.acm.org/citation.cfm?id=1776434.1776436.

[179] Jiankun Hu, Xinghuo Yu, Dong Qiu, and Hsiao-Hwa Chen. A simple and efficient hidden Markov model scheme for host-based anomaly intrusion detection. *Network*, 23(1):42–47, 2009.

[180] Weiming Hu, Wei Hu, and Steve Maybank. Adaboost-based algorithm for network intrusion detection. *Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 38(2):577–583, 2008. URL http://dx.doi.org/10.1109/TSMCB.2007.914695.

[181] Yuehui Chen, Ajith Abraham, and Bo Yang. Hybrid flexible neural-tree-based intrusion detection systems. *International Journal of Intelligent Systems*, 22(4): 337–352, 2007. URL http://dx.doi.org/10.1002/int.v22:4.

[182] Jakob Fritz, Corrado Leita, and Michalis Polychronakis. Server-Side Code Injection Attacks: A Historical Perspective. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013. URL http://dx.doi.org/10.1007/978-3-642-41284-4_3.

[183] Periklis Akritidis, Evangelos P Markatos, Michalis Polychronakis, and Kostas Anagnostakis. Stride: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of the IFIP 20th International Information Security Conference (ISC)*, 2005. URL https://doi.org/10.1007/0-387-25660-1_25.

[184] Stig Andersson, Andrew Clark, George Mohay, Bradley Schatz, and Jacob Zimmermann. A framework for detecting network-based code injection attacks targeting Windows and UNIX. In *Proceedings of the 21th Annual Computer Security Applications Conference (ACSAC)*, 2005. URL http://dx.doi.org/10.1109/CSAC.2005.5.

[185] Michalis Polychronakis, Kostas G Anagnostakis, and Evangelos P Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the 3rd International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2006.

[186] Ramkumar Chinchani and Eric Van Den Berg. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006. URL http://dx.doi.org/10.1007/11663812_15.

[187] Qinghua Zhang, Douglas S Reeves, Peng Ning, and S Purushothaman Iyer. Analyzing network traffic to detect self-decrypting exploit code. In *Proceedings of the 2nd Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2007. URL http://doi.acm.org/10.1145/1229285.1229291.

[188] Sherly Abraham and InduShobha Chengalur-Smith. An overview of social engineering malware: Trends, tactics, and implications. *Technology in Society*, 32(3): 183–196, 2010. URL https://doi.org/10.1016/j.techsoc.2010.07.001.

[189] Mordechai Guri, Matan Monitz, and Yuval Elovici. Bridging the air gap between isolated networks and mobile phones in a practical cyber-attack. *ACM Trans. Intell. Syst. Technol.*, 8(4):50:1–50:25, 2017. ISSN 2157-6904. URL http://doi.acm.org/10.1145/2870641.

[190] Niels Provos et al. A Virtual Honeypot Framework. In *Proceedings of the 13th USENIX Security Symposium*, 2004. URL http://dl.acm.org/citation.cfm?id=1251375.1251376.

[191] J. Nazario. PhoneyC: a virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more (LEET)*, 2009.

[192] Christian Seifert, Ian Welch, and Peter Komisarczuk. Honeyc-the low-interaction client honeypot. *Proceedings of the New Zealand Computer Science Research Student Conference (NZCSRCS)*, 2007.

[193] Christian Seifert, Ian Welch, and Peter Komisarczuk. Identification of Malicious Web Pages with Static Heuristics. In *Australasian Telecommunication Networks and Applications Conference (ATNAC)*, 2008. URL https://doi.org/10.1109/ATNAC.2008.4783302.

[194] Steve Morgan. CYBERSECURITY MARKET REPORT - Q2 2017. https://cybersecurityventures.com/cybersecurity-market-report/, 2017. [Online; accessed February 2018].

[195] Ganapathy Krishnan and Scott Oyler. Method and system for injecting new code into existing application code, 2000. US Patent 6,141,698.

[196] A. Ghizzoni. Method for injecting code into another process, 2004. US Patent 6,698,016.

[197] Ray Heasman and James Baker. System and method for foiling code-injection attacks in a computing device, 2006. US Patent App. 11/521,866.

[198] Weon Il Jin, Hwan Joon Kim, Eun Ah Kim, and Gyungho Lee. Apparatus and method for detecting a code injection attack, 2013. US Patent 8,615,806.

[199] James Mensch, Jerry Hauck, and Ronnie Misra. Run-time code injection to perform checks, 2013. US Patent 8,375,369.

[200] James Yun. Method and apparatus for monitoring code injection into a process executing on a computer, 2013. US Patent 8,612,995.

[201] Seung Bae Park. Code injection prevention, 2014. US Patent 8,769,672.

[202] Crowdstrike. Crowdstrike Falcon. https://www.crowdstrike.com/products/, 2017. [Online; accessed February 2018].

[203] Cylance. CylancePROTECT. https://www.cylance.com/en_us/products/our-products/protect.html, 2017. [Online; accessed February 2018].

[204] Barkly. Barkly Endpoint Protection Platform. https://www.barkly.com/product, 2017. [Online; accessed February 2018].

[205] Microsoft. Windows Defender Advanced Threat Protection. https://www.microsoft.com/en-us/windowsforbusiness/windows-atp, 2017. [Online; accessed February 2018].

[206] Christian Seifert, Genghis Karimov, Mathieu Letourneau. Uncovering cross-process injection with Windows Defender ATP. https://blogs.technet.microsoft.com/mmpc/2017/03/08/uncovering-cross-process-injection-with-windows-defender-atp/, 2017. [Online; accessed February 2018].

[207] John Lundgren. Detecting stealthier cross-process injection techniques with Windows Defender ATP: Process hollowing and atom bombing. https://blogs.technet.microsoft.com/mmpc/2017/07/12/detecting-stealthier-cross-process-injection-techniques-with-windows-defender-atp-process-hollowing-and-atom-bombing/, 2017. [Online; accessed February 2018].

[208] Christian Seifert. Detecting reflective DLL loading with Windows Defender ATP. https://blogs.technet.microsoft.com/mmpc/2017/11/13/detecting-reflective-dll-loading-with-windows-defender-atp/, 2017. [Online; accessed February 2018].

[209] Symantec. Internet Security Threat Report 2013, Volume 18. http://www.symantec.com/about/news/resources/press_kits/detail.jsp?pkid=istr-18, 2013. [Online; accessed February 2018].

[210] Qing Dong, Runze Zhao, and Nianzhong Sun. Oldboot.B: the hiding tricks used by bootkit on Android. http://blogs.360.cn/360mobile/2014/04/02/analysis_of_oldboot_b_en/, 2014. [Online; accessed February 2018].

[211] Uroburos - highly complex espionage software with russian roots. https://public.gdatasoftware.com/Web/Content/INT/Blog/2014/02_2014/documents/GData_Uroburos_RedPaper_EN_v1.pdf, 2014. [Online; accessed February 2018].

[212] Dennis Schwarz. Citadel's Man-In-The-Firefox: An Implementation Walk-Through. http://www.arbornetworks.com/threats/citadel.pdf, 2013. [Online; accessed February 2018].

[213] Josh Grunzweig. The Dexter Malware: Getting Your Hands Dirty. http://blog.spiderlabs.com/2012/12/the-dexter-malware-getting-your-hands-dirty.html, 2012. [Online; accessed February 2018].

[214] James Wyke. Zeroaccess. http://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/ZeroAccess.pdf, 2012. [Online; accessed February 2018].

[215] Mark W. Eichin and Jon A. Rochlis. With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988. In *Proceedings of the 10th Symposium on Security and Privacy (S&P)*, 1989.

[216] Fei Su, Zhaowen Lin, and Yan Ma. Effects of firewall on worm propagation. In *International Conference on Communications Technology and Applications (ICCTA)*, 2009. URL https://doi.org/10.1109/ICCOMTA.2009.5349051.

[217] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Protecting Web Services from Remote Exploit Code: A Static Analysis Approach. In *Proceedings of the 17th international conference on World Wide Web (WWW)*, 2008. URL http://doi.acm.org/10.1145/1367497.1367695.

[218] Kyumin Lee, James Caverlee, and Steve Webb. Uncovering Social Spammers: Social Honeypots + Machine Learning. In *Proceedings of the 33rd international Conference on Research and development in information retrieval (SIGIR)*, 2010. URL http://doi.acm.org/10.1145/1835449.1835522.

[219] M. Karresand. Separating trojan horses, viruses, and worms - a proposed taxonomy of software weapons. In *Information Assurance Workshop*, 2003. URL https://doi.org/10.1109/SMCSIA.2003.1232411.

[220] Ilfak Guilfanov. Windows WMF Metafile Vulnerability HotFix. http://www.hexblog.com/?p=21, 2005. [Online; accessed February 2018].

[221] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[222] Aaron Portnoy. MindshaRE: Debugging via Code Injection with Python. [Online; accessed February 2018], 2011.

[223] More human than human – Flame's code injection techniques. http://www.cert.pl/news/5874, 2012. [Online; accessed February 2018].

[224] GovCERT.ch. APT Case RUAG. https://www.melani.admin.ch/dam/melani/en/dokumente/2016/technical%20report%20ruag.pdf.download.pdf/Report_Ruag-Espionage-Case.pdf, 2016. [Online; accessed February 2018].

[225] Joxean Koret and Elias Bachaalany. *The Antivirus Hacker's Handbook*. Wiley Publishing, 1st edition, 2015. ISBN 9781119028758.

[226] Marco Giuliani. ZeroAccess – an advanced kernel mode rootkit. http://botnetlegalnotice.com/zeroaccess/files/Ex_12_Decl_Anselmi.pdf, 2011. [Online; accessed February 2018].

[227] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, 5th edition, 2011. ISBN 9780132143011.

[228] Kins source code. https://github.com/nyx0/KINS, 2015. [Online; accessed February 2018].

[229] John Von Neumann. First Draft of a Report on the EDVAC. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[230] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012. URL http://doi.acm.org/10.1145/2133375.2133377.

[231] Ralf Hund, Thorsten Holz, and Felix C Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium*, 2009. URL http://dl.acm.org/citation.cfm?id=1855768.1855792.

[232] Aleksandr Matrosov. Win32/Gapz: steps of evolution. http://www.welivesecurity.com/2012/12/27/win32gapz-steps-of-evolution/, 2012. [Online; accessed February 2018].

[233] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. deRop: removing return-oriented programming from malware. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, 2011. URL http://doi.acm.org/10.1145/2076732.2076784.

[234] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. Dynamic Hooks: Hiding Control Flow Changes Within Non-control Data. In *Proceedings of the 23rd USENIX Security Symposium*, 2014. URL http://dl.acm.org/citation.cfm?id=2671225.2671277.

[235] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. Persistent data-only malware: Function Hooks without Code. In *Proceedings of the 21st Symposium on Network and Distributed System Security (NDSS)*, 2014. URL https://doi.org/10.1007/978-3-319-26362-5_9.

[236] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. Omniunpack: Fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.124.2886.

[237] Maarten van Dantzig, Danny Heppener, Frank Ruiz, Yonathan Klijnsma, Yun Zheng Hu, Erik de Jong, Krijn de Mik, and Lennart Haagsma. Ponmocup: A giant hiding in the shadows. https://foxitsecurity.files.wordpress.com/2015/12/foxit-whitepaper_ponmocup_1_1.pdf, 2015. [Online; accessed February 2018].

[238] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P)*, 2012. URL http://dx.doi.org/10.1109/SP.2012.16.

[239] Antonio Nappa, M. Zubair Rafique, and Juan Caballero. The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations. *International Journal of Information Security*, pages 1–19, 2014. URL http://dx.doi.org/10.1007/s10207-014-0248-7.

[240] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your

pocket. *Proceedings of the 21st Symposium on Network and Distributed System Security (NDSS)*, 2014.

[241] Microsoft. Microsoft Malware Classification Challenge (BIG 2015). https://www.kaggle.com/c/malware-classification, 2015. [Online; accessed February 2018].

[242] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2017. URL https://doi.org/10.1007/978-3-319-60876-1_12.

[243] Federico Maggi, Andrea Bellini, Guido Salvaneschi, and Stefano Zanero. Finding Non-trivial Malware Naming Inconsistencies. In *Proceedings of 7th International Conference on Information Systems Security*, 2011. URL http://dx.doi.org/10.1007/978-3-642-25560-1_10.

[244] Christian Rossow, Christian J. Dietrich, Christian Kreibich, Chris Grier, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *Proceedings of the 33rd Symposium on Security and Privacy (S&P)* , 2012. URL https://doi.org/10.1109/SP.2012.14.

[245] M. Bailey, J. Oberheide, J. Andersen, Z. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*. 2007. URL http://dl.acm.org/citation.cfm?id=1776434.1776449.

[246] Virus Sign: Automated Malware Analysis System. http://www.virussign.com/vsamas.html. [Online; accessed February 2018].

[247] VirusTotal. https://www.virustotal.com, . [Online; accessed February 2018].

[248] Aziz Mohaisen and Omar Alrawi. AV-Meter: An Evaluation of Antivirus Scans and Labels. In *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2014. URL https://doi.org/10.1007/978-3-319-08509-8_7.

[249] Peter Kleissner. Sality: 2003 - today. https://www.botconf.eu/wp-content/uploads/2015/12/OK-P18-Kleissner-Sality.pdf, 2015. [Online; accessed February 2018].

[250] Microsoft News Center. Microsoft, the FBI, Europol and industry partners disrupt the notorious ZeroAccess botnet. http://news.microsoft.com/2013/12/05/microsoft-the-fbi-europol-and-industry-partners-disrupt-the-notorious-zeroaccess-botnet/, 2013. [Online; accessed February 2018].

[251] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[252] Denis Bueno, Kevin J Compton, Karem A Sakallah, and Michael Bailey. Detecting traditional packers, decisively. In *Proceedings of the 16th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2013. URL https://doi.org/10.1007/978-3-642-41284-4_10.

[253] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting Environment-sensitive Malware. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2011. URL http://dx.doi.org/10.1007/978-3-642-23644-0_18.

[254] Olivier Ferrand. How to detect the Cuckoo Sandbox and to Strengthen it? *Journal of Computer Virology and Hacking Techniques*, 11(1):51–58, 2015. URL https://doi.org/10.1007/s11416-014-0224-9.

[255] Paul Schofield and Udi Yavo. PowerLoaderEx. https://github.com/BreakingMalware/PowerLoaderEx, 2015. [Online; accessed February 2018].

[256] Brian Bartholomew and Juan Andres Guerrero-Saade. Wave your false flags! deception tactics muddying attribution in targeted attacks. In *Virus Bulletin Conference*, 2016.

[257] Cuckoo Sandbox. http://www.cuckoosandbox.org. [Online; accessed February 2018].

[258] Kuba Udymowski-Grecki. Defeating Antivirus Real-time Protection From The Inside. https://breakdev.org/defeating-antivirus-real-time-protection-from-the-inside/, 2016. [Online; accessed February 2018].

[259] Microsoft. Control flow guard. https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065%28v=vs.85%29.aspx, . [Online; accessed February 2018].

[260] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008. ISBN 0470128720.

[261] Terrehon Bowden, Bodo Bauer, Jorge Nerin, Shen Feng, and Stefani Seibold. The /proc Filesystem. https://www.kernel.org/doc/Documentation/filesystems/proc.txt. [Online; accessed February 2018].

[262] Thomas Barabosch. beemaster-complementary-material. https://github.com/tbarabosch/beemaster-complementary-material, 2017. [Online; accessed February 2018].

[263] Tyler Colgan. linux-inject. https://github.com/gaffe23/linux-inject, 2015. [Online; accessed February 2018].

[264] Dan Staples. linux-injector. https://github.com/dismantl/linux-injector, 2015. [Online; accessed February 2018].

[265] Thomas Barabosch. 1001-injects. https://github.com/tbarabosch, 2017. [Online; accessed February 2018].

[266] Mark Russinovich and David A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009. ISBN 0735625301, 9780735625303.

[267] scikit-learn. http://scikit-learn.org, 2016. [Online; accessed February 2018].

[268] Pedro Domingos. A Few Useful Things to Know About Machine Learning. *Communications of the ACM*, 55(10):78–87, 2012. URL http://doi.acm.org/10.1145/2347736.2347755.

[269] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in computer virology*, 7(2), 2011. URL https://doi.org/10.1007/s11416-010-0141-5.

[270] Mamoun Alazab, Sitalakshmi Venkatraman, Paul Watters, and Moutaz Alazab. Zero-day malware detection based on supervised learning algorithms of api call signatures. In *Proceedings of the 9th Australasian Data Mining Conference*, pages 171–182. Australian Computer Society, Inc., 2011. URL http://dl.acm.org/citation.cfm?id=2483628.2483648.

[271] IOActive. Reversal and Analysis of Zeus and SpyEye Banking Trojans. https://www.ioactive.com/pdfs/ZeusSpyEyeBankingTrojanAnalysis.pdf, 2012. [Online; accessed February 2018].

[272] Daniel Kusswurm. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Apress, 2014. ISBN 1484200659, 9781484200650.

[273] Or Safran. Gozi Banking Trojan Upgrades Build to Inject Into Windows 10 Edge Browser. https://securityintelligence.com/gozi-banking-trojan-upgrades-build-to-inject-into-windows-10-edge-browser/, 2016. [Online; accessed February 2018].

[274] avast blog Threat Intelligence Team. Andromeda under the microscope. https://blog.avast.com/andromeda-under-the-microscope, 2016. [Online; accessed February 2018].

[275] Dominik Reichel. 2016 Updates to Shifu Banking Trojan. http://researchcenter.paloaltonetworks.com/2017/01/unit42-2016-updates-shifu-banking-trojan/, 2016. [Online; accessed February 2018].

[276] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th Conference on Computer and Communications Security (CCS)*, 2003. URL http://doi.acm.org/10.1145/948109.948149.

[277] Brendan Dolan-Gavitt. The vad tree: A process-eye view of physical memory. *Digital Investigation*, 4, 2007. URL http://dx.doi.org/10.1016/j.diin.2007.06.008.

[278] Michael Hale Ligh, Steven Adair, Blake Hartstein, and Matthew Richard. *Malware Analyst's Cookbook and DVD: Tools And Techniques For Fighting Malicious Code*. Wiley Publishing, Inc., 1 edition, 2011.

[279] Josh Kaufman. google-10000-english. https://github.com/first20hours/google-10000-english, 2016. [Online; accessed February 2018].

[280] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene Selection for Cancer Classification Using Support Vector Machines. *Journal of Machine learning*, 46(1-3):389–422, 2002. URL http://dx.doi.org/10.1023/A:1012487302797.

[281] Robin Genuer, Jean-Michel Poggi, and Christine Tuleau-Malot. Variable Selection Using Random Forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010. URL http://dx.doi.org/10.1016/j.patrec.2010.03.014.

[282] Brandon Rohrer. How to choose algorithms for Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/documentation/articles/machine-learning-algorithm-choice/, 2015. [Online; accessed February 2018].

[283] Rich Caruana and Alexandru Niculescu-mizil. An Empirical Comparison of Supervised Learning Algorithms. In *Proceedings of 23rd International Conference on Machine learning (ICML)*, 2006. URL http://doi.acm.org/10.1145/1143844.1143865.

[284] David D Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *European Conference on Machine Learning*, 1998. URL http://dl.acm.org/citation.cfm?id=645326.649711.

[285] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European Conference on Computational Learning Theory*, 1995. URL http://dx.doi.org/10.1006/jcss.1997.1504.

[286] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.29.9093.

[287] Barreno, Marco and Nelson, Blaine and Sears, Russell and Joseph, Anthony D. and Tygar, J. D. Can Machine Learning Be Secure? In *Proceedings of the ACM Symposium on Information, Computer and Communications Security 2006 (AsiaCCS)*, 2006. URL http://doi.acm.org/10.1145/1128817.1128824.

[288] Portableapps.com. http://portableapps.com/. [Online; accessed February 2018].

[289] Alberto Ortega. Pafish. https://github.com/a0rtega/pafish. [Online; accessed February 2018].

[290] YARA. https://plusvic.github.io/yara/. [Online; accessed February 2018].

[291] Kent Griffin, Scott Schneider, Xin Hu, and Tzi-Cker Chiueh. Automatic generation of string signatures for malware detection. In *International Conference on Recent Advances in Intrusion Detection (RAID)*, 2009. URL http://dx.doi.org/10.1007/978-3-642-04342-0_6.

[292] Sam Roweis and Lawrence Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500), 2000. URL http://doi.org/10.1126/science.290.5500.2323.

[293] Joshua Tenenbaum, Vin De Silva, and John Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500), 2000.

[294] Ulisses M Braga-Neto and Edward R Dougherty. Is cross-validation valid for small-sample microarray classification? *Journal of Bioinformatics*, 20(3):374–380, 2004. URL http://dx.doi.org/10.1093/bioinformatics/btg419.

[295] Rushi Longadge and Snehalata Dongre. Class imbalance problem in data mining: Review. *arXiv preprint arXiv:1305.1707*, 2013. URL http://citeseerx.ist.psu.edu/viewdoc/citations;jsessionid=65724F3F3CE0C2C6A134E58E29DD4ADC?doi=10.1.1.300.7601.

[296] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012. URL http://dl.acm.org/citation.cfm?id=2188385.2188395.

[297] Philip J Clark and Francis C Evans. Distance to nearest neighbor as a measure of spatial relationships in populations. *Ecology*, 35(4):445–453, 1954. URL https://www.doi.org/10.2307/1931034.

[298] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification, 2003.

[299] Danny Kim, Amir Majlesi-Kupaei, Julien Roy, Kapil Anand, Khaled ElWazeer, Daniel Buettner, and Rajeev Barua. Dynodet: Detecting dynamic obfuscation in malware. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017. URL https://doi.org/10.1007/978-3-319-60876-1_5.

[300] Matt Dahl. Sakula Reloaded. https://www.crowdstrike.com/blog/sakula-reloaded/, 2015. [Online; accessed February 2018].

[301] hasherezade. Rokku Ransomware shows possible link with Chimera. https://blog.malwarebytes.com/threat-analysis/2016/04/rokku-ransomware/, 2016. [Online; accessed February 2018].

[302] Github Help. About Stars. https://help.github.com/articles/about-stars/, 2017. [Online; accessed February 2018].

[303] Magal Baz and Or Safran. Dridex's Cold War: Enter AtomBombing. https://securityintelligence.com/dridexs-cold-war-enter-atombombing/, 2017. [Online; accessed February 2018].

[304] Ziyun Zhu and Tudor Dumitras. Featuresmith: Automatically engineering features for malware detection by mining the security literature. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016. URL http://doi.acm.org/10.1145/2976749.2978304.

[305] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[306] Kyoung-Su Oh and Keechul Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37(6), 2004. doi: https://doi.org/10.1016/j.patcog.2004.01.013.

[307] Baidu Research. http://research.baidu.com, 2017. [Online; accessed February 2018].

[308] Facebook AI Research (FAIR). https://research.facebook.com/ai/, 2017. [Online; accessed February 2018].

[309] Google deepmind. https://deepmind.com, 2017. [Online; accessed February 2018].

[310] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015. URL https://doi.org/10.1109/MALWARE.2015.7413680.

[311] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence*, 2016.

[312] Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering convolutional networks for malware classification and analysis. In *International Joint Conference on Neural Networks (IJCNN)*, 2017.

[313] Saso Džeroski and Bernard Ženko. Is Combining Classifiers with Stacking Better than Selecting the Best One? *Journal of Machine Learning*, 54(3):255–273, 2004. URL http://dx.doi.org/10.1023/B:MACH.0000015881.36452.6e.

[314] Joseph Sill, Gábor Takács, Lester W. Mackey, and David Lin. Feature-Weighted Linear Stacking. 2009. URL https://arxiv.org/abs/0911.0460.

[315] Daniel Plohmann. Malpedia. https://malpedia.caad.fkie.fraunhofer.de. [Online; accessed February 2018].

[316] Microsoft. Microsoft security essentials. http://windows.microsoft.com/en-us/windows/security-essentials-download, . [Online; accessed February 2018].