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

Thomas Barabosch, Niklas Bergmann, Adrian Dombeck, and Elmar Padilla

Fraunhofer FKIE, Zanderstrasse 5, 53177 Bonn, Germany
firstname.lastname@fkie.fraunhofer.de

Abstract. Malware predominantly employs code injections, which allow to run code in the trusted context of another process. This enables malware, for instance, to secretly operate or to intercept critical information. It is crucial for analysts to quickly detect injected code. While there are systems to detect code injections in memory dumps, they suffer from unsatisfying detection rates or their detection granularity is too coarse. In this paper, we present *Quincy* to overcome these drawbacks. It employs 38 features commonly associated with code injections to classify memory regions. We implemented *Quincy* for Windows XP, 7 and 10 and compared it to the current state of the art, *Volatility*'s *malfind* as well as *hollowfind*. For this sake, we created a high quality data set consisting of 102 current representatives of code injecting malware families. *Quincy* improves significantly upon both approaches, with up to 19.49% more true positives and a decrease in false positives by up to 94,76%.

Keywords: Malware;Memory Forensics;Host-Based Code Injection Attacks;Machine Learning

1 Introduction

Malware families implement many different behaviors such as form grabbing, information leakage, persistence or code injections. Host-Based Code Injection Attacks (HBCIA) are a family-inherit technique utilized to execute code in a trusted context of another process. There are two processes involved: the attacker process P_a and the victim process P_v, which both run on the same system. P_a injects code from its own process space into the one of P_v. Subsequently, P_a triggers the execution of this code within P_v. HBCIAs allow malware, for instance, to intercept critical information within a browser or to hide from antivirus software. A study indicates that almost two thirds of recent malware samples utilize this technique [7]. Amongst others, this includes prevalent families like *Dridex*, *Rovnix*, *Tinba* and *Zeus*. This points out the relevance of HBCIAs and renders them an interesting and valuable topic to investigate in order to detect and mitigate malware in general.

Analysts face HBCIAs on a daily basis. Forensic analysts are confronted with memory dumps of unknown systems and in case of a malware infection without an initial sample. Malware analysts continue to integrate forensic analyses in

their work flow due to the improvement of memory forensic frameworks like *Volatility* [27]. In both cases, a fast and accurate initial detection of malicious code in a memory dump is crucial. There are systems such as *Volatility*'s *malfind* [27], *hollowfind* [20] and *Membrane* [24] that support the detection of HBCIAs in memory dumps. However, they suffer from major drawbacks. Whereas *malfind* suffers from a high false positive rate, *hollowfind* detects only a subgroup of all relevant types of HBCIAs. For example, *malfind* fails to detect *Ponmocup*, driving one of the biggest botnets out in the wild [26]. It fails because it only considers two features: the access property of memory regions as well as the hiding of libraries. In contrast to *malfind* and *hollowfind*, *Membrane* is limited to a coarse grain detection, i.e. it detects infected processes instead of the actual malicious regions within a process. Although this reduces the size of the hay stack to search in, it does not pinpoint the malware exactly: for instance, the process *Explorer* contains 554 memory regions yielding 405 MB of data on an idling Windows 10 system.

In this paper, we overcome the limitations of the aforementioned systems by presenting *Quincy*. Its detection heuristic is based on 38 features from seven categories commonly associated with injected code. They include, among others, memory region permissions, memory region sparseness and the presence of shellcode. *Quincy* embeds these features in a vector space and classifies consecutive memory pages (in the following just *memory regions*) as either malicious or benign. We implemented *Quincy* for three Windows versions and released it as a *Volatility* plugin on our website [5]. Our evaluation with a set of 102 current malware families and 1794 benign programs shows that our system has a higher detection rate with only few false positives (up to 94,76%) and more true positives (up to 19,49%) when compared to *malfind* and *hollowfind*.

The contributions of this paper can be summarized as follows:

(I) **A novel approach for HBCIA detection**
We propose a fully-automated system to detect Host-Based Code Injection Attacks in memory dumps. *Quincy* has the idea of platform-independence in mind and hence focuses on concepts found among all modern multitasking operating systems. Our approach is based on supervised machine learning and utilizes a combination of 38 features to detect HBCIAs. This allows it to significantly improve upon the state of the art *malfind*.

(II) **Implementation and evaluation**
We implemented *Quincy* and released it as a *Volatility* plugin on our website [5]. We evaluated it in a systematic evaluation with current real world malware families and goodware. In addition, we compared it to *malfind* as well as *hollowfind* to prove that it significantly improves upon them.

(III) **Creation and publication of our data set**
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 crafted YARA signatures for each

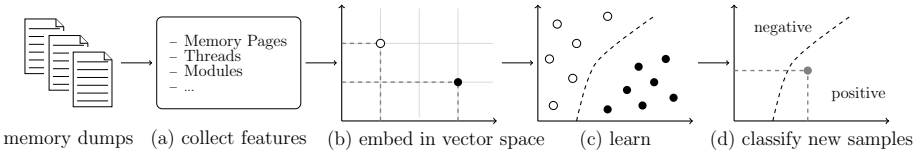

Fig. 1: The four phases of *Quincy*: it receives dumps with labeled memory regions as input. Then, it extracts 38 HBCIA-related features, which are organized in seven categories (a). Subsequently, it selects valuable features and embeds them in a multi-dimensional vector space (b). It then induces a binary tree-based classifier (c). Finally, it can classify previously unseen memory regions (d).

family to verify a successful infection and to ensure a precise ground truth. We share this data set on our website [5].

2 Quincy

In this section, we present *Quincy*, our approach to detect HBCIAs in memory dumps. At first, we give an overview of its architecture. The system consists of the following phases: feature extraction, feature selection, embedding of these features into a vector space, learning and classification. Subsequently, we describe each of these phases in detail.

2.1 Overview

Figure 1 sketches *Quincy*'s work flow: first, it receives memory dumps as input and closes the semantic gap, i.e. the gap between the binary representation of data in a memory dump and its meaning to the operating system. Internally, this is done by the memory forensic framework *Volatility* [27].

Second, it extracts low-level information, including processes, threads and memory regions. The features are based on this low-level information. They are closely related to HBCIAs such as memory region permissions, dynamic API resolving and the presence of shellcode. *Quincy* extracts these features for each region in a memory dump. A memory region is a set of consecutive pages within a process. Whereas the typically page size is four kilobyte on x86, a memory region consisting of many pages may have a size of several megabytes. *Virtual Address Descriptor* (VADs) is the term for a memory region on Windows. Note that such regions may be shared between processes, e.g. system libraries that are mapped with *EXECUTE_WRITECOPY* permissions on Windows.

Third, *Quincy* embeds these features for every memory region in a multi-dimensional vector space. Fourth, it induces a binary classifier. As a result, it can classify previously unseen memory region as either malicious or benign.

2.2 Feature Extraction

The following sections describe the 38 features organized in seven categories that *Quincy* may employ for classification. We engineered these features based upon domain knowledge in the fields of malware analysis and memory forensics. Table 1 summarizes the features. Later, we conduct a feature selection to discard less valuable features (see Section 4.1). Thereby, we create a feature set for each operating system. This optimizes our detection rate while minimizing the resources required.

(1) API System interaction such as network communication and file access can only be accomplished through the operating system via syscalls. On most operating systems high-level APIs are available that are more comfortable to use. This included Windows. These APIs are an important keystone in the malware analysis process: the presence of certain function calls allows to draw conclusions about the behavior of a binary. For instance, we can deduce from a call to *CreateRemoteThread* that an *HBCIA* is likely to occur.

The feature $api_{general\_api\_strings}$ checks for the presence of API calls in general by scanning for common string prefixes such as *Create*, *Get* or *Open*. This enables us to differentiate memory regions that might communicate with the OS and the ones that might not. As a consequence this detects regions hosting executable files. The feature api_{hbcias} explicitly scans for a set of API calls that are related to HBCIAs such as *CreateToolhelp32Snapshot* and *ZwSetContextThread*.

Since the presence of API calls lessens the analyst's burden, malware authors obfuscate API names. Hence, they deobfuscate them just in time and dynamically resolve the pointers to the API code. This can also be done via a set of special functions. The feature $api_{dynamic\_loading}$ checks for the presence of such functions like *LoadLibrary* and *GetProcAddress*. A more sophisticated method is to manually resolve APIs by enumerating all libraries that are mapped into the process space. This requires access to process data structure, e.g. the *Process Environment Block* (PEB) on Windows. The feature $api_{hashing}$ searches for code patterns that access such data structures to detect code that implements api hashing.

(2) Binary Executable programs and libraries are building blocks of each process. They are also known as *modules*, which typically pose as a memory region. Programs and libraries match formats like the *Portable Executable* (PE) standard on Windows or the *ELF* standard on Linux and have a well-defined header. The following features interpret header structures in memory regions in case they are available.

The feature $binary_{has\_header}$ checks if a memory region starts with a well-known header. However, malware may overwrite its header to impede its analysis. The feature $binary_{wiped\_header}$ covers this case by checking for a zeroed-out beginning of a memory region that is followed by code. Although benign programs come in the form of a stand-alone executable or a library, malware often

Overview of *Quincy*'s features			
category	**feature**	**rank**	**description**
(1) **API**	dynamic_loading	29/26/16	presence of dynamic loading APIs
	general_api_strings	08/11/13	common API call prefixes
	hashing	09/10/17	code fragments related to API hashing
(2) **binary**	exports	30/19/33	exports API calls
	has_header	23/21/20	starts with a header
	imports	35/33/28	imports API calls
	is_dynamic_library	32/20/27	has been loaded dynamically
	is_module	16/13/22	registered module known to the OS
	is_pe_or_dll	14/16/10	a PE executable or shared library
	wiped_header	37/34/36	executable header has been wiped
(3) **code**	functions	10/08/18	common assembler function prologues
	hooks	04/05/12	memory region contains code hooks
	indirect_calls	05/03/05	ratio of indirect calls to all calls
	indirect_jumps	12/04/07	ratio of indirect jumps to all jumps
	shellcode	01/15/11	shellcode patterns
(4) **cryptography**	cipher	33/29/30	constants of ciphers
	encoding	26/23/21	constants of encoding schemes
	hashing	28/20/25	constants of hashing algorithms
(5) **countermeasure detection**	debugger	17/18/29	strings and code patterns to detect debuggers
	sandbox	22/27/15	strings and code patterns to detect sandboxes
	vm	36/36/35	strings and code patterns to detect virtual machines
(6) **memory**	embedded_exe	38/38/38	embedded executable after header
	english_strings	27/35/23	strings of Google's top 1000 English search terms
	high_entropy_areas	07/06/06	areas of high entropy
	is_heap	34/32/32	memory region is a heap
	is_sparse	03/01/02	ratio of zero bytes
	mapped	15/37/37	corresponds to a memory mapped file
	network_strings	06/07/14	strings related to networking
	persistence	24/30/19	strings related to persistence
	private	18/14/08	tagged as private memory
	protection	13/17/01	protection of memory region
	tag	20/09/05	tagged by allocation functions
	threads	11/12/09	count of threads originated in memory region
	victim_strings	19/31/26	names of typical HBCIA victims
(7) **trojan**	banking	25/28/31	strings related to online banking
	cookies	21/23/24	strings related to cookie stealing
	credentials	31/24/34	strings related to credential stealing

Table 1: Summary of the 38 features utilized by *Quincy*. The categories and features within them are alphabetically arranged. The rank of a feature is based on its importance determined in the feature selection on Windows XP/7/10. Note that the final models do not employ all features (see Section 4.6).

injects shellcode into its victim processes. The feature $binary_{is\_pe\_or\_dll}$ checks if a memory region is a stand-alone executable or library by reading a field of the corresponding header. Benign executables and libraries are either loaded at process start or dynamically during runtime with the help of the OS, which keeps track of these modules. The feature $binary_{is\_module}$ encodes if a memory region is registered as an official module by parsing the PEB. Malware obfuscates its API usage and therefore imports few or none API functions. The feature $binary_{imports}$ encodes whether or not a memory region imports such functions.

Malware may also inject entire libraries into victim processes. System libraries export up to several hundreds functions. In contrast, malware may only export a handful of functions, if any. The feature $binary_{exports}$ checks if a region has exported functions.

(3) Code The following features scrutinize assembly code properties of a memory region. We assume that every meaningful program is split into several units of code, which is reflected by low-level assembly functions. Therefore, the feature $code_{functions}$ searches for patterns of common function prologues in memory regions. We assume these byte sequences to indicate code presence. Malware families like *GozNym* do not inject executable modules such as standalone executables and libraries but rather shellcode. On execution, this position-independent code has to determine its current address in memory to act. The feature $code_{shellcode}$ scans memory regions for code patterns that determine their position in memory. For example, it considers patterns like a call to the next instruction, followed by a pop to a register, which determines the current address in memory.

Due to position-independence and obfuscation reasons, malware contains significantly more branches with dynamically calculated targets in relation to direct calls and jumps. The features $code_{indirect\_calls}$ and $code_{indirect\_jumps}$ describe the ratio of indirect calls/jumps to all calls/jumps. The feature $code_{hooks}$ searches for code hooks that point from one memory region into another region. The presence of such hooks may reveal, for example, the presence of banking trojans that hook libraries in browsers to intercept banking credentials. On the downside, searching for hooks in memory dumps is computational expensive. For instance, *Volatility*'s *apihooks* may take up to a couple of minutes to scan a memory image. Therefore, we opted to scan memory regions for strings related to hooking of browser APIs functions like *Firefox*'s *Netscape Portable Runtime* (NSPR), which are commonly hooked by code-injecting banking trojans.

(4) Cryptography Malware may try to hide its presence and communication by extensive use of cryptography, e.g. files are encrypted with *AES*, network traffic is encoded with *Base64* or network packets are hashed with *SHA256*. Usually malware does not rely on external libraries like Microsoft's Cryptographic API, but rather statically links the cryptographic algorithms into its binary in order to increase analysis costs. Features of this category look for constants or strings related to prominent encryption ($crypto_{cipher}$), encoding ($crypto_{encoding}$) and hashing algorithms ($crypto_{hashing}$).

(5) Countermeasure Detection Malware authors want to postpone analysis as long as possible. Therefore, many malware families impede their analysis by including countermeasure techniques. We distinguish between three types: first, the feature $counter_{debugger}$ checks for traces of anti-debugging techniques that aim at manual analysis. These traces include code fragments, e.g. accessing

the *beingDebugged* flag of the process space, and strings related to malware analysis tools such as debuggers and process inspectors. Second, the feature $counter_{sandbox}$ checks for the presence of certain sandbox related strings, e.g. such as *Anubis* or *Cuckoo*. Third, the feature $counter_{vm}$ scans for strings and code fragments that detect virtual machines, which are commonly employed in malware analysis. For example, malware can detect *VirtualBox* VMs due to its default MAC address prefix *0x080027*.

(6) Memory The following features focus on the properties of memory regions themselves. They are arranged in three subcategories.

Statistical features: many memory regions are sparse, i.e. they have a high ratio of zero bytes. In contrast, memory regions of binaries are more densely filled with data. Therefore, the feature $memory_{is\_sparse}$ measures the ratio of zero bytes to find nearly empty regions.

Lyda et al. [18] proposed the entropy of data to detect compressed or encrypted data. *Quincy* leverages entropy analysis to detect areas of high entropy. We chose the area size to be four kilobyte as the typical page size on the Intel x86 architecture and the entropy threshold to be 6.5 as suggested by Lyda et al. [18]. The feature $memory_{high\_entropy\_areas}$ encodes the percentage of high entropy areas within a memory region.

Memory region properties: this subcategory considers mostly flags assigned to a memory region by the operating system: heap flag ($memory_{is\_heap}$), its protections ($memory_{protection}$) such as readable, writable or executable, memory mapped file flag ($memory_{mapped}$), private memory flag ($memory_{private}$) and memory allocation function tag ($memory_{tag}$). Furthermore, the feature $memory_{threads}$ determines if any thread has been started within a region.

Strings: whereas strings may be obfuscated on hard disk, there are surprisingly many strings in memory. This also holds for malware employing executable packing. The feature $memory_{english\_strings}$ matches a word frequency list of the thousand most frequent search terms on *Google* consisting of more than three characters. We assume that this may help to identify rather benign regions. The feature $memory_{network\_strings}$ detects memory regions that contain networking vocabulary such as *HTTP* or *POST* since network communication is an integral part of today's malware. HBCIA-employing malware prefers certain victim processes [7]. The feature $memory_{victim\_strings}$ searches for victim names in regions such as *explorer.exe* or *svchost.exe* to identify memory regions of HBCIA-employing malware. The feature $memory_{persistence}$ detects strings related to persistence, e.g. the Windows registry key $\ldots\backslash CurrentVersion\backslash Run$ to find code that may have achieved persistence on the system.

(7) Trojan One reason to inject code into another process is to intercept information. Therefore, code injections are especially important to trojans like *GozNym*, *Xswkit* or *KINS*. Since the main objective of banking trojans is to divert money in banking sessions, the feature $trojans_{banking}$ scans every memory

region for a comprehensive list of financial vocabulary and bank names. Furthermore, they target cookies and general credentials such as *Facebook* or *LinkedIn* accounts. The two features $trojans_{cookies}$ and $trojans_{credentials}$ scan for strings related to cookies and credentials correspondingly.

2.3 Feature Selection and Embedding in Vector Space

Before utilizing machine learning, we select a set of appropriate features and embed them in a vector space without standardization. We employ machine learning algorithms that are unaffected by varying feature scales (tree-based algorithms, see next section). Most features are of binary nature, e.g. the feature $memory_{embedded\_executable}$. However, there are also continuous features such as $memory_{high\_entropy\_areas}$ and $code_{indirect\_calls}$.

Initially, the vector space has 38 dimensions. We carry out a *recursive feature elimination* (RFE) as proposed by Guyon et al. [16]. RFE employs an external estimator that weights features based on their importance. It is recursively trained with decreasing feature sets, where it pruns the weakest feature in each iteration. For this sake, we employ *Random Forests* as external estimator as proposed by Genuer et al. [13].

2.4 Learning and Classification

Quincy learns a model to classify memory regions either as malicious or benign. There are several classes of machine learning algorithms for classification problems such as *Support Vector Machines*, *Logistic Regression* and *Decision Tree-based algorithms*. Tree-based algorithms pose several advantages including the comprehensibility of predictions, the simplicity of the algorithms and the minimal effort required in data preparation. Therefore, we opt for *Decision Tree-based algorithms*, considering *CART-Decision Trees* [10], *Random Forests* [9], *Extremely Randomized Trees* [14], *AdaBoost* [11] and *GradientBoosting* [12].

2.5 Implementation

We implemented *Quincy* in *Python*. It leverages the memory forensic framework *Volatility* [27] to extract features and *scikit-learn* [3] to learn. Our implementation analyzes all Windows NT versions from Windows XP onwards.

To speed up the analysis process, Quincy copies memory images to a RAM disk and conducts the feature extraction in memory. Hence, the read speed of the machine's memory is crucial to the general runtime. Furthermore, *Quincy*'s feature extraction is single-threaded. Therefore, we expect further speed up by parallelizing the feature extraction.

3 Data Set Creation

We follow the advices of Rossow et al. [25], based on the fact that an evaluation requires a comprehensive data set. This section describes how we created the data

data set	year	publication	R1	R2	R3	R4
cwsandbox	2007	[29]	✗	✗	✓	✗
Malicia	2013	[21]	✗	✗	✓	✗
Malware Classification Challenge	2015	[19]	✓	✗	✓	✗

Table 2: Matching of publicly available data sets to our four requirements to an evaluation data set discussed in Section 3.1

set for our evaluation and what kind of data it comprises. First, we describe the considered binaries and how we generated memory dumps from them for the evaluation. Next, we show how we properly labeled the memory regions to ensure a reliable ground truth. Finally, we conduct an initial data analysis of the extracted data.

3.1 Data Set

We require an evaluation data set to contain:

R1 a considerable amount of HBCIA-employing malware families
R2 recent malware families
R3 only Windows malware
R4 goodware programs to estimate false positives

The data set should contain a considerable amount of different families to evaluate the systems with different code injection techniques (**R1**). Next, we want to ensure that our evaluation results are valid for recent malware (**R2**) that runs on the prevalent target Microsoft Windows (**R3**). Finally, the set should contain goodware programs to estimate false positives (**R4**). Table 2 matches these four requirements to three publicly available data sets. *cwsandbox* as well as *malicia* contain only older malware strains and hence violate requirement **R2**. *Malware Classification Challenge* consists of a considerable amount of samples. However, they belong to less than ten families, not all of which employing code injections. Since none of these data sets matches our requirements, we opted to create our own and contribute it to the research community.

We considered 1794 benign as well as 102 malicious binaries and generated a memory dump for each of them. Memory dumps of malware contain benign and malicious memory regions, while dumps generated with benign binaries are assumed to contain only benign regions. In the following sections, we describe which binaries we considered in detail.

Benign Binaries Benign binaries comprise software included in Windows and other widespread programs. For this sake, we extracted programs from the system directory of Windows XP, 7 and 10. Additionally, we collected binaries of widespread programs from an archive of portable freeware applications [1]. In total, we collected 1794 benign binaries. However, the programs that we were

able to execute varied among Windows versions. Some programs were not compatible with each version. Moreover, we did not execute system programs of one version on another to ensure compatibility. A list of all benign binaries and their hashes is provided on our website [5].

Malicious Binaries Our set of malicious binaries consist of 102 representatives of HBCIA employing malware families. Barabosch et al. [6] showed that HBCIAs are an inherent malware family feature, i.e. it is unlikely to change between versions and variants of a family. Therefore, it is sufficient to consider one representative per family. Note that this minimizes family specific overfitting by focusing on the employed HBCIA techniques. We gathered family representatives over the last months. On the one hand, we consulted IT security blogs, e.g. of antivirus companies, that carried out in-depth analysis of malware families. On the other hand, we included families that we internally analyzed at our institute. Later, we manually verified the HBCIA capability of the obtained families (see Section 3.3). The set of malicious families contains a wide range of current malware that represent today's threat landscape, for instance, viruses (*Sality*), banking trojans (*Xswkit*), spamming bots (*Cutwail*) and droppers (*Nymaim*). Some samples are not compatible with every Windows version, therefore the number of executable families varies. We share the malicious binaries on our website [5].

3.2 Creation of Memory Dumps

We generated memory dumps for Windows XP SP3, Windows 7 SP1 and Windows 10. We automated the memory dump generation process with a tool, which is based on the virtualization software *VirtualBox* [22]. First, it creates an ISO image containing the sample. Then, it starts the virtual machine in a predefined state and mounts the ISO image on the virtual CD Drive. The guest system runs a script, which executes the sample with administrator privileges. We grant each sample two minutes to initialize, which is a common timeout of sandboxing systems. At the end, it dumps the memory state of the virtual machine to a file.

The virtual machines were not connected to the Internet during the infection, since no command and control server communication was required. They were hardened against several virtual machine detection techniques, since malware may be environment sensitive [17]. First, we utilized the tool *Pafish* [23] to find ways to detect our VMs. Subsequently, we hardened detection points, e.g. by removing strings of the hypervisor from the registry. Note that sometimes hardening is not feasible. For instance, fixing subtle differences between a real x86 CPU and the implementation provided by *VirtualBox* are out of scope for this work.

3.3 Establishing a Ground Truth

A proper labeling of the data set is essential for a reliable evaluation of our model and comparison with the state of the art *malfind* [27] and *hollowfind* [20]. We

OS	binaries		memory regions	
	benign	**malicious**	**benign**	**malicious**
Windows XP	1205	71	2729563	398
Windows 7	1264	72	5306368	319
Windows 10	977	73	7266226	710
Total (unique)	1794	102	15302157	1427

Table 3: Data distribution of benign and malicious binaries as well as benign and malicious memory regions for all three considered Windows versions.

established our labeling as follows: we assumed all memory regions of goodware dumps as benign. We can not make a similar assumption for malware dumps: the regions may be malicious or benign. Therefore, we opt to employ *YARA* signatures [2] to reliably detect malicious artifacts in memory dumps. For this sake, we manually reverse engineered the malware families and wrote signatures for each of them. Even though some preliminary work on automatic signature generation exists [15], it is limited to static signatures. However, malware usually is packed, i.e. the original binary and the executed code in memory significantly differ.

We estimated the detection rates of *malfind* and *hollowfind* by interpreting their results as follows: in case they did not mention a memory region then it was labeled as benign. In the other case, it was labeled as malicious.

3.4 Initial Data Analysis

We conducted an initial analysis of the extracted data to get a first impression. According to Table 3, it exhibits a skewed class distribution. The benign binaries outnumber the malicious binaries by an order of magnitude. Benign binaries are easier to access than properly labeled representatives of HBCIA-employing malware. Thereby, the distribution is even more skewed in the case of benign and malicious memory regions, because there are typically more benign processes and hence more benign memory regions than infected processes and malicious regions, respectively. Nevertheless, we argue that this is exactly the haystack scenario that detection systems face in the wild.

4 Model Evaluation

We select features and conduct an optimization and evaluation of our model in Section 4.1 - 4.3. This is followed by the evaluation of our optimized model and a comparison with *malfind* as well as *hollowfind* in Section 4.4. Then, we conduct a temporal analysis to estimate how well *Quincy* detects future malware families in Section 4.5. Section 4.6 summarizes the final models that we learned on the whole data sets. Finally, we discuss evasion strategies for *Quincy* in Section 4.7.

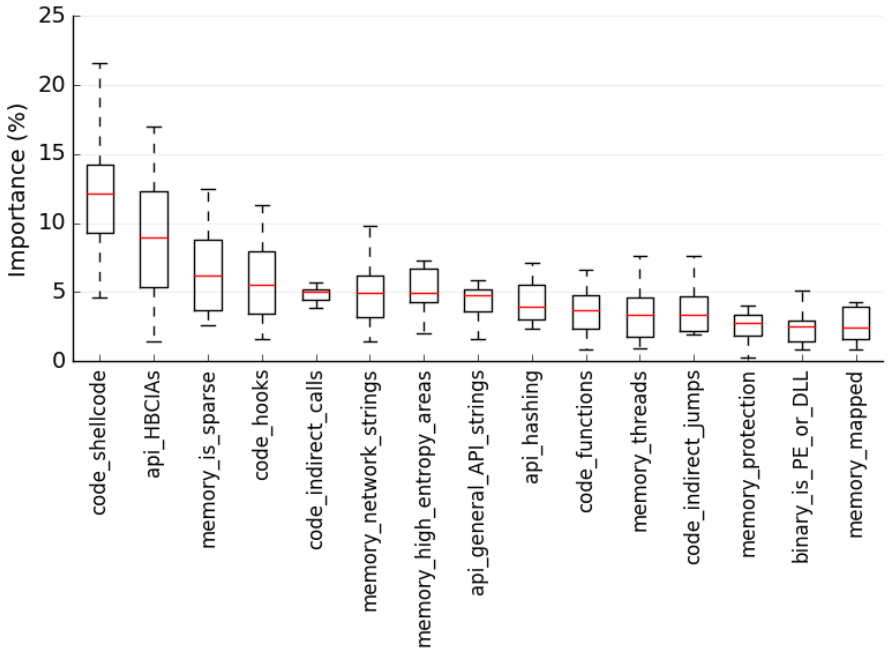

Fig. 2: Relative feature importance in percent for the top 15 features on Windows XP. We obtained these values during the recursive feature elimination phase in the model learning stage. They are based on the estimation of the *Random Forests*.

4.1 Methodology of the Model Evaluation

We did the following ten times for each data set of the three Windows versions, in a cross-validation loop in order to cope with variance.

At first, we randomly split the whole data set consisting of the malicious and benign memory regions into two sets. The first set is the training set d_{train} and the second set is the validation set $d_{validation}$. d_{train} was utilized in model training and model optimization, whereas $d_{validation}$ was exclusively utilized to estimate the final performance of our optimized model. Therefore, we evaluated our optimized model on unseen data to estimate its potential for generalization. Please note that we split the data set such that the malicious regions of one family were either in d_{train} or in $d_{validation}$ but never in both sets. This ensured that our model did not face malicious regions of one family in training and validation.

We showed that the class distribution is heavily skewed in our initial data analysis (see Section 3.4). Machine learning algorithms may perform poorly and misclassify many minority class instances due to optimizing the overall accuracy and hence shifting their focus to the majority class. Therefore, we treated the

two classes separately. We split malicious regions with a ratio of 60%/40% into d_{train} and $d_{validation}$ and benign regions with a ratio of 10%/90%. On the one hand, this ensured that there are sufficient malicious samples in d_{train}. On the other hand, this added noise in form of benign regions to $d_{validation}$. Noise that *Quincy*, *malfind* and *hollowfind* are confronted with in the real world.

We selected the optimal set of features on d_{train} using *Recursive Feature Elimination* (RFE) with Random Forests. Afterwards, we trained and optimized several tree-based models on d_{train} for comparing them later. The model optimization took place on the whole set of d_{train} and was carried out in form of a randomized grid search. Finally, we evaluated the optimized models together with *malfind* and *hollowfind* on $d_{validation}$.

4.2 Feature Selection

The feature selection took place on d_{train} using *Recursive Feature Elimination* (RFE) with Random Forests. Figure 2 shows the relative feature importance on Windows XP. Note that the results are similar on Windows 7 and Windows 10. Table 1 lists the ranking of the features for the three operating systems.

There are only a few features that significantly contribute to the model with an average of more than 5%. The top three features are $code_{shellcode}$, $memory_{contains\_HBCIA\_strings}$ and $memory_{is\_sparse}$. Whereas the first two features directly aim at detecting malicious memory regions, the third feature detects close to empty and hence probably benign memory regions. Surprisingly, $memory_{contains\_HBCIA\_strings}$ performs well even though it scans for strings. Furthermore, all features of the categories *code* and API as well as one half of the features of *memory* are contained within the top 15 features. They cover the two integral parts of a code injection: the injected code and its execution context. In contrast, the two categories $trojan$ and *cryptography* do not perform as expected. The assumptions on which these two categories are based did not hold. Whereas in the case of *cryptography* we assumed that malware prefers to statically link cryptographic algorithms, in the case of $trojan$ we assumed that data theft related vocabulary is present in many malware strains.

There are few features that have an importance of less than one percent. They are either rare cases like $memory_{embedded\_executable}$ and $binary_{wiped\_header}$ or they are common in regular programs like $crypto_{cipher}$. The feature $memory_{protection}$ on which *malfind* heavily relies on is only of medium importance to our model.

4.3 Optimization of Hyperparameters

The optimization of hyperparameters is an important step towards an optimal model. They are defined outside of the machine learning algorithm, e.g. maximal tree depth or the number of base estimators in a learning ensemble. We opted for a randomized grid search to optimize the hyperparameters of our tree-based models. It does not search over every grid point of the hyperparameter space, instead it randomly samples grid points and evaluates the model with them. Bergstra et al. [8] showed that randomized grid search is more effective than

algorithm	trees	learning rate	max. features	tree depth
AdaBoost [11]	[10,400]	[0.1,1.0]	-	-
CART [10]	1	-	$\sqrt{\|f\|}$, $\|f\|$	[3,12] + ∞
Extremely Randomized Trees [14]	[10,400]	-	$\sqrt{\|f\|}$, $\|f\|$	-
GradientBoosting [12]	[10,400]	[0.1,1.0]	-	[4,8]
Random Forest [9]	[10,400]	-	$\sqrt{\|f\|}$, $\|f\|$	-

Table 4: Hyperparameters to optimize and their value ranges for the five tree-based machine learning algorithms. f denotes the total number of features.

exhaustive grid search as it converges to a close-to-optimum solution at a high rate. We sampled 64 grid points in total as suggested by Bergstra et al. [8] and conducted a 10-fold cross validation for every sampled grid point. We chose the best performing parameters for our final model. The evaluation metric was the area under the receiver operating characteristic curve (ROC AUC score).

We considered five algorithms: *CART-Decision Trees*, *Random Forests*, *Extremely Randomized Trees*, *AdaBoost* and *GradientBoosting*. All of them have several hyperparameters: some of these hyperparameters affect the whole ensemble like the learning rate and some of them affect the individual trees like the maximal tree depth. Optimizing all of these parameters is computationally expensive. Therefore, we decided to optimize only the most significant four hyperparameters and set the others to *scikit-learn*'s [3] default values. Table 4 lists the hyperparameters and their respective value ranges.

4.4 Model Evaluation

After having trained and optimized our models, we evaluated their final performance on unseen data to estimate how well they generalize. In addition, we compared them to the state of the art approach *malfind* in version 2.5 [27] as well as *hollowfind* [20], the Volatility plugin contest winner of 2016. *malfind* extensively focuses on memory region related features like memory region permissions. *hollowfind* detects process hollowing by finding discrepancies in process data structures. For an exact description of *malfind* and *hollowfind* see Section 5.

We evaluated *Quincy* with five tree-based machine learning algorithms. The following holds true for all three operating systems: the standard decision tree algorithm *CART* yields more true positives than *malfind* but comes with an order of magnitude more false positives. *AdaBoost* and *GradientBoosting* detect less false positives than *malfind*, however they also detect less malicious regions, resulting in an overall worse performance. These two algorithms exhibit far better results than non-boosting algorithms on the training data, which lets us conclude that they most likely overfit. The two best performing algorithms are *Random Forests* and *Extremely Randomized Trees*. Both algorithms are bagging-based. They dominate *malfind* in all cases, with *Extremely Randomized Trees* being the

Windows	Quincy			malfind			hollowfind		
	AUC	**TP**	**FP**	**AUC**	**TP**	**FP**	**AUC**	**TP**	**FP**
XP	93.8%	149.1	813.5	90.2%	137.9	15538.3	52.24%	7.7	1306.4
7	88.5%	94.4	547.3	80.4%	76.0	7488.0	52.70%	6.6	9176.8
10	84.4%	187.3	1828.5	81.9%	175.6	3672.0	51.57%	8.8	110.1

Table 5: Final data of the evaluation of *Quincy* with *Extremely Randomized Trees*, *malfind* and *hollowfind* on $d_{validation}$: Area Under Curve (AUC), True Positives (TP) and False Positives (FP).

most successful. *Quincy* and *malfind* dominated *hollowfind* in all cases, which showed only slightly better performance than throwing a coin.

Table 5 lists the final results of *Quincy* with *Extremely Randomized Trees*, *malfind* and *hollowfind*. All values represent the mean of the 10-fold cross validation. Our system dominates *malfind* when comparing their area under the ROC curve. Its highest score is 93,8% on Windows XP. The greatest difference between their AUC scores can be observed on Windows 7 with 8,09%. Both *Quincy* and *malfind* outperform *hollowfind*, which only detects process hollowing. This is a special case of HBCIAs.

Quincy with *Extremely Randomized Trees* detects more true positives than *malfind* with up to 19,49% on Windows 7. Since *Quincy* incorporates one of *malfind*'s two features, we assumed equal performance at least. However, our system considers more features and detects more malicious memory regions. In contrast to *malfind*, it detects, for instance, malware families like Ponmocup and Dridex, which inject libraries into their victim processes.

Quincy has also less false positives than *malfind* with up to 94,76% on Windows XP. *malfind* considers every non-empty memory area with *RWX* permissions as malicious. Malware authors might forget to cover their traces or the architecture (e.g. shellcode) demands these permissions. This allows *malfind* at least to partially detect a family. However, once these permissions are adjusted well (i.e. only *RX* permissions are set), *Quincy* outperforms *malfind* due to its comprehensive set of other features. False positives of our approach include programs like *Dropbox Portable* that exhibit similar signs like the malicious regions: high entropy areas (probably due to packing), presence of shellcode to determine its position in memory, *RWX* memory permissions and extensive use of cryptography. This results in similar features like of malicious regions. They are therefore falsely classified as malicious. Another observation is that falsely assumed malicious regions decrease with more modern Windows versions. Therefore, *malfind* false positive rate decreases, however our system benefits from this as well.

Table 6 shows the family detection and family completeness. We consider a family *detected* if one approach detects at least one of the family's memory regions. Note that a malware infection may result in many distinctive memory regions distributed over several processes. A detection is considered *complete* when all memory regions are detected. On average, *Quincy* detects more malware families on Windows 7 than *malfind*. Even though both exhibit a similar family

Windows	families	Quincy		malfind		hollowfind	
		detection	complete	detection	complete	detection	complete
XP	29	24.3	20.9	24.4	19.5	4.7	0
7	29	26.4	18.1	24.3	11.9	4.6	0
10	30	23.7	18.1	23.6	15.2	4.9	0

Table 6: Family detection and family completeness of *Quincy* with *Extremely Randomized Trees*, *malfind* and *hollowfind* on $d_{validation}$.

detection on Windows XP and Windows 10, *malfind* often just detects small malicious regions with *RWX* permissions but misses on the main module of the malware family. While this may confirm an infection, it does not yield the main payload responsible for the infection, which is essential to carry out further investigations. This assumption is also supported by the family completeness. On average, our system completely detects more families since it does not solely focus on memory region permissions. *hollowfind* detected only the families that employ process hollowing and none of them completely. Overall our proposed model dominates the other approaches in all evaluation metrics.

4.5 Temporal Evaluation

After evaluating *Quincy* with *malfind* and *hollowfind* and showing its superiority, we conducted a temporal evaluation of these three systems. The objective was to evaluate how well they perform in a temporal setting, i.e. training them on older, historical data and evaluating them on recent data. This evaluation proceeds similar to the general one, limited to one iteration of the cross validation loop. The chronological order is based on a family's first occurrence in the wild. For this sake, we queried *VirusTotal* [4] to arrange the malware in chronological order and split it with a 60%/40% ratio.

Figure 3 shows the final performance as ROC curve of the three approaches averaged over all three operating systems. It documents the superiority of *Quincy* in the temporal evaluation comparing an AUC score of 90.9% versus 87.6% of *malfind* and 54.3% of *hollowfind*. An interesting finding is that the ways of injecting code, e.g. hollowing processes by using memory mapped files or creating a remote thread, do not substantially differ in the two data sets, meaning that newer families exhibit similar injection traces as older families. An explanation may be that malware authors tend to copy from each other.

4.6 Final Models

We precomputed three models based on the full data sets, which we distribute with *Quincy*'s source code [5]. Therefore, we chose *Extremely Randomized Trees* as learning algorithm based on its performance. Moreover, we carried out a feature selection using *RFE* with *Random Forests* and optimized the hyperparameters *number of trees* and *maximal number of considered features*. Table 7 presents the number of selected features and the final hyperparameters.

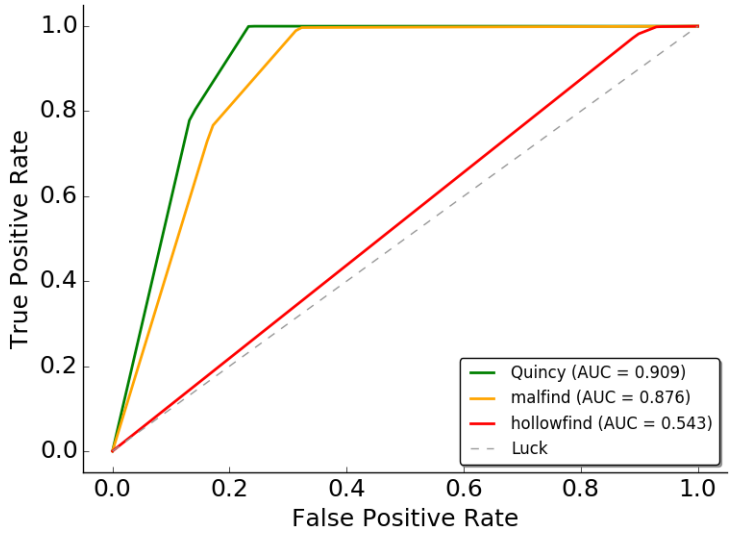

Fig. 3: Final performance of *Quincy* (green), *malfind* (orange) and *hollowfind* (red) in temporal evaluation illustrated as ROC curves averaged over all three evaluated operating systems (XP, 7, 10)

4.7 Discussion of Evasion

As with every detection system, an adversary can try to understand and circumvent its detection heuristic. However, we utilize 38 features from seven categories. These categories scrutinize several aspects of a memory region, from its memory properties to its embedded code. An adversary may try to circumvent some features, e.g. by bloating the code with zeros to resemble a sparse region, but there are still other features like $code_{functions}$ that would indicate that the region may contain relevant code. The number of different features and their correlations increases the challenge to circumvent our system, when compared to other systems like *malfind* or *hollowfind*.

But there are HBCIAs that might not be detectable during a post-mortem memory dump analysis. Several operating systems offer the possibility to load arbitrary libraries into a process during its creation, e.g. *AppInit DLLs* on Windows. If malware employs such means then detection may fail for several reasons. Foremost, the library has been loaded by the system loader in the same fashion as a regular library. Such modules have therefore the same permissions or they are listed in the same data structures as regular system libraries. In the case of the absence of other indicators these injections are especially difficult to detect.

Windows	number of features	number of trees	maximal features
XP	27	445	$\sqrt{\|f\|}$
7	34	475	$\sqrt{\|f\|}$
10	23	435	$\sqrt{\|f\|}$

Table 7: The number of optimal features and hyperparameters for *Quincy* with *Extremely Randomized Trees* on the evaluated operating systems. The number of features describes the amount of features that were selected during the feature selection. The two hyperparameters were selected during a randomized grid search on the whole data set of each operating system.

5 Related Work

There are four systems allowing forensic detection of HBCIAs in memory dumps, which are closely related to our approach: *malfind* [27], *hollowfind* [20], *Membrane* [24] and *Hashtest* [28]. All of them are based on the memory forensic framework *Volatility*. There are public implementations of each except *Membrane*. Table 8 compares them to *Quincy*.

Malfind Hale Ligh proposed the current state of the art *malfind* [27]. It implements a combination of two features to classify memory regions. At first, it marks entirely empty regions as benign. Pages with *RWX* protections and unlinked libraries (from the PEB) are marked as malicious. Furthermore, it detects wiped PE headers in *RWX*-protected memory regions. The remaining regions are assumed to be benign. Its detection heuristic may be completely circumvented by not utilizing *RWX* permissions and not unlinking libraries. Lassalle proposed *malfinddeep* [27], an improvement to *malfind* that utilizes whitelisting of memory regions based on ssdeep hashes. We did not evaluate *malfinddeep* since there is no official whitelist available. Our work significantly improves upon *malfind*, as shown in the evaluation. *Quincy* considers a superset of *malfind*'s features adding many more in order to decrease false positives, increase true positives and render evasion more difficult.

Hollowfind The volatility plugin *hollowfind* [20] detects process hollowing, which is a code injection technique, replacing code of a legitimate process and manipulating the initial thread to execute malicious code. The behavior of the malware blends in a trusted process, e.g. *svchost.exe*. *hollowfind* detects process hollowing by comparing two process management data structures for discrepancies. It considers the *Process Environment Block* (PEB), which amongst others list the loaded modules with their paths. Furthermore, it considers a data structure in kernel space (VAD structure), which contains information about the modules' paths. If *hollowfind* finds a discrepancy for a process, then it assumes it to be hollowed out and outputs its memory regions with *RWX* protection like *malfind* does. Its heuristic may be circumvented by not using process hollowing

approach	heuristic	features	granularity	compatibility
malfind [27]	rule-based	2	memory region	XP +
hollowfind [20]	rule-based	2	memory region	XP +
Membrane [24]	*Random Forrest*	23/28*	process	XP and 7
Hashtest [28]	hash comparison	1	memory region	XP and 7
Quincy	*Extremely Randomized Trees*	38	memory region	XP +

Table 8: Comparison of related approaches. *XP +* implies that the approach runs on every Windows version since Windows XP. * *Membrane* considers 23 features on Windows XP and 28 features on Windows 7.

or by removing the discrepancies from the PEB. Overall, the scope of *hollowfind* is narrower than *Quincy*'s. Our system detects HBCIAs in general, a superset including process hollowing.

Membrane Pek et al. propose *Membrane* [24]. It reconstructs low-level memory paging information of Windows's software memory management unit (MMU) and leverages this information to detect HBCIAs. Based on domain knowledge, they identified 23 features on Windows XP and 28 features on Windows 7 and applied a *Random Forest* classifier to detect HBCIAs on process-granularity.

There are overlappings 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 is finer. Whereas *Membrane* detects HBCIAs on process-granularity, *Quincy* detects them on memory region-granularity. Therefore, a direct comparison between them is not possible. Second, Pek et al.'s approach is very prone to noise. Their results drastically decline from 98% accuracy on Windows XP to 73% on Windows 7. We assume that on Windows 10 this problem gets even worse since the noise level increases with every Windows version as our evaluation showed. Third, they implemented their approach for two older Windows versions (XP and 7). Quincy is not limited to a certain Windows version, hence it also runs on the latest version. Fourth, *Membrane* is 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 requires tedious reverse engineering.

Hashtest White et al. present *Hashtest* [28]. They detect HBCIAs by hashing memory regions and subsequently searching for these hashes in a previously built hash database. This reduces the amount of memory regions to analyze. *Quincy* does not rely on whitelisting. Therefore, our approach generalizes better and can deal with previously unseen data.

6 Conclusion

Host-Based Code Injection Attacks (HBCIAs) play an important role in modern malware with at least two thirds employing them [7]. A fast initial detection of injected malicious code in memory dumps is crucial. Therefore, we presented *Quincy*, a system for detecting these attacks on memory region basis. It is based on supervised machine learning, utilizing 38 HBCIA-related features, selecting the optimal feature set, embedding these features in a vector space and training a tree-based model for classification. The evaluation showed that *Extremely Randomized Trees* fit especially well to the problem.

We evaluated our system on Windows XP, 7 and 10. For this purpose, we created a data set according to the best practices and published the data set online. We generated memory dumps for more than one thousand benign and malicious binaries and created a comprehensive data set of benign and malicious memory regions based on a sound ground truth. Based on this data set, we evaluated *Quincy* and compared it to the current state of the art *malfind* and *hollowfind*. Our results show that *Quincy* significantly improves upon them. It has less false positives as well as more true positives and dominates the other approaches on all three considered Windows versions. Finally, we enable practitioners to take advantage of our findings by publishing our implementation [5].

7 Acknowledgment

The final publication is available at Springer via `https://doi.org/10.1007/978-3-319-60876-1_10`.

References

1. The Portable Freeware Collection. `http://www.portablefreeware.com`. Last accessed: August 21, 2017.
2. YARA. `https://plusvic.github.io/yara/`. Last accessed: August 21, 2017.
3. scikit-learn. `http://scikit-learn.org`, 2016. Last accessed: August 21, 2017.
4. VirusTotal. `https://www.virustotal.com`, Last access: August 21, 2017.
5. T. Barabosch, N. Bergmann, A. Dombeck, and E. Padilla. Quincy Project Site. `https://net.cs.uni-bonn.de/wg/cs/staff/thomas-barabosch/`. Last accessed: August 21, 2017.
6. T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla. Bee Master: Detecting Host-Based Code Injection Attacks. *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2014.
7. T. Barabosch and E. Gerhards-Padilla. Host-Based Code Injection Attacks: A Popular Technique Used By Malware. *Malicious and Unwanted Software (MALCON)*, 2014.
8. J. Bergstra and Y. Bengio. Random Search for Hyper-parameter Optimization. *Journal of Machine Learning Research (JMLR)*, 2012.
9. L. Breiman. Random forests. *Machine learning*, 45, 2001.
10. L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.

11. Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*. Springer, 1995.
12. J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, 2001.
13. R. Genuer, J.-M. Poggi, and C. Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225–2236, 2010.
14. Geurts, Pierre and Ernst, Damien and Wehenkel, Louis. Extremely randomized trees. *Machine learning*, 63, 2006.
15. K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection (RAID)*, 2009.
16. Guyon, Isabelle and Weston, Jason and Barnhill, Stephen and Vapnik, Vladimir. Gene selection for cancer classification using support vector machines. *Machine learning*, 2002.
17. M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection (RAID)*, 2011.
18. R. Lyda and J. Hamrock. Using Entropy Analysis to Find Encrypted and Packed Malware. *Security and Privacy (S&P)*, 2007.
19. Microsoft. Microsoft Malware Classification Challenge (BIG 2015). `https://www.kaggle.com/c/malware-classification`, 2015, Last access: August 21, 2017.
20. K. A. Monnappa. Detecting Deceptive Process Hollowing Techniques Usind Hollowfind Volatility Plugin. `https://cysinfo.com/detecting-deceptive-hollowing-techniques/`, 2016. Last accessed: August 21, 2017.
21. A. Nappa, M. Z. Rafique, and J. Caballero. The MALICIA Dataset: Identification and Analysis of Drive-by Download Operations. *International Journal of Information Security*, pages 1–19, June 2014.
22. Oracle. VirtualBox. `https://www.virtualbox.org`. Last accessed: August 21, 2017.
23. A. Ortega. Pafish. `https://github.com/a0rtega/pafish`. Last accessed: August 21, 2017.
24. G. Pék, Z. Lázár, Z. Várnagy, M. Félegyházi, and L. Buttyán. Membrane: A Posteriori Detection of Malicious Code Loading by Memory Paging Analysis. In *European Symposium on Research in Computer Security (ESORICS)*. Springer, 2016.
25. C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, and M. Van Steen. Prudent practices for designing malware experiments: Status quo and outlook. In *Security and Privacy (SP)*, 2012.
26. M. van Dantzig, D. Heppener, Y. K. Frank Ruiz, Y. Z. Hu, E. de Jong, K. de Mik, and L. Haagsma. Ponmocup - A giant hiding in the shadows. `https://foxitsecurity.files.wordpress.com/2015/12/foxit-whitepaper_ponmocup_1_1.pdf`, 2015. Last accessed: August 21, 2017.
27. Volatility Foundation. The Volatility Framework. `http://www.volatilityfoundation.org`, 2015. Last accessed: August 21, 2017.
28. A. White, B. Schatz, and E. Foo. Integrity verification of user space code. *Digital Forensic Research Workshop (DFRWS)*, 2013.
29. C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *Proceedings of the 28th Symposium on Security and Privacy (S&P)*, 2007.