An Intelligent Malware Classification Framework

by

Elaheh Biglar Beigi Samani

Bachelor of Information Technology, IUT, 2010

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of
Faculty of Computer Science

Supervisor(s): Ali Ghorbani, Ph.D, Faculty of Computer Science
Examining Board: Rodney H. Cooper, Professor, Faculty of Computer Science, Chair
Natalia Stakhanova, Ph.D., Faculty of Computer Science
Brent Petersen, Ph.D., Department of Electrical and Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

September, 2015

©Elaheh Biglar Beigi Samani, 2015

Abstract

Malicious software or malware has risen to become a primary source of most of the attacks taking place across the Internet over the last decades. This prevalence of new malware, for which signatures are not available, along with the challenge of anti-malware software to keep up with the continuous stream of new malware, has made the adoption of classification/-clustering approaches necessary. Machine-learning methods have been excessively applied to classify or cluster malware into families, based on different features derived from static or dynamic review of the malware. While these approaches demonstrate promise, they are themselves subject to a growing array of countermeasures. In this work, we propose a framework to enhance the traditional machine learning-based classification by utilizing high-level domain knowledge. We outline major behaviours of Windows malware from an analyst's point of view and provide possible methods (rules) to extract them from the output of static and dynamic analysis tools. We also take advantage of memory forensics to extract other stealthy aspects of an executable, which otherwise remain undetected. Our comparative experimental results with the state-of-the-art malware classification approaches, confirm the effectiveness of our framework by an average classification accuracy of 81%, while leaving only 0.5% of samples unlabeled.

Dedication

I dedicate my thesis work to my family. A special feeling of gratitude to my loving husband, Hossein Hadian, whose words of encouragement and push for tenacity ring in my ears, and to my parents, who never left my side and are very special.

Acknowledgements

I would like to express my sincere appreciation to my supervisor, Dr. Ali Ghorbani, for his guidance, encouragement and constant support through this thesis. I also acknowledge the help and support of my friends and colleagues at the Information Security Centre of Excellence, who always helped me keep my spirits high during hard times.

Table of Contents

Abstract **ii**

Dedication **iii**

Acknowledgments **iv**

Table of Contents **viii**

List of Tables **ix**

List of Figures **xi**

Abbreviations **xii**

1 Introduction **1**

1.1 Introduction 1

1.2 Summary of Contributions 4

1.3 Thesis Organization 5

2 Literature Review **6**

2.1 Static Malware Classification 6

2.2 Dynamic Malware Classification 7

2.3 Hybrid Malware Classification 8

2.4 Behaviours-based Malware Detection 9

2.4.1 Replication and Propagation . . . 10
2.4.2 Malicious Arguments . . . 11
2.4.3 Persistancy . . . 11
2.4.4 Hooking and DLL injection . . . 12
2.4.5 Environment Fingerprinting . . . 15
2.4.6 Process Anomalies . . . 15
2.4.7 Registry Access Anomalies . . . 16
2.4.8 Sensitive Data Access or Modification . . . 17
2.5 Concluding Remarks . . . 17

3 Proposed Framework for Intelligent Malware Classification 18
3.1 Overview . . . 18
3.2 Analysis . . . 20
3.3 Knowledge Correlator . . . 20
3.3.1 Level-one Rules . . . 21
3.3.2 Level-two Rules . . . 24
3.3.3 Level-three Rules . . . 25
3.4 Text to Binary Converter . . . 26
3.5 Classification . . . 28
3.5.0.1 Prototype Extraction . . . 28
3.5.0.2 Clustering of Prototypes . . . 29
3.5.0.3 Classification . . . 29
3.5.0.4 Profiles Distance . . . 31
3.6 Concluding Remarks . . . 32

4 Memory Forensics 33
4.1 Overview . . . 33

4.2 Process Anomalies 33
4.3 DLL Anomalies 34
4.4 Attacking IDT/GDT 35
4.5 New Drivers/Devices 35
4.6 Anomalous Timers 37
4.7 Hidden drivers 38
4.8 Thread Anomalies 38
4.9 SSDT Anomalies 39
4.10 API Hooks 39
4.11 Malicious Callbacks 41
4.12 Conclusion 43

5 Implementation **44**

5.0.1 Configuration 44
5.0.2 ReportCreator 46
5.0.3 AnalysisReport 46
5.0.4 KnowledgeCorrelator 47
5.0.5 LevelOneParser 47
5.0.6 LevelTwoParser 47
5.0.7 LevelThreeParser 49
5.0.8 CMIST 51
5.0.9 TesxToNumeric 51
5.0.10 MachineLearning 52
5.0.11 BehaviouralProfile 52
5.1 Conclusion 53

6 Experiments and Results **54**

6.1 Datasets and Labeling 54
6.2 System Calibration 56
6.3 Comparative Analysis 59
6.3.1 Comparative Evaluation with Malheur 59
6.3.2 Comparative Evaluation with Sally 64
6.3.3 Run-time and Memory Requirements 65
6.4 Concluding Remarks 66

7 Conclusions and Future Work **67**
7.1 Conclusions 67
7.2 Future Work 68

Bibliography **79**

Appendixes **79**

A Interesting function calls **80**

B List of common auto-run registry keys **91**

C Windows core processes **95**

Vita

List of Tables

2.1 Summary of malware detection/classification approaches 9

3.1 CWSandbox and Cuckoo comparison 23

6.1 Statistics on types of malware in the dataset. 56
6.2 Size of samples/profiles in MB. 56
6.3 Details on samples rejected in both approaches. 63
6.4 Evaluation results 65

B.1 Auto start registry keys 94

C.1 Known facts about Windows OS core processes 97

List of Figures

3.1 General overview of the proposed malware classification framework 19

3.2 A snippet of behavioral profile - output of level-one rules for OneStepSearch 21

3.3 Example of system call generated by CWSandbox [56] 22

3.4 Example of MIST representation for system call *load_dll* [56] 22

3.5 Example of MIST definition for system call *loadDll* 24

3.6 A snippet of behavioral profile - output of level-two rules for OneStepSearch 25

3.7 A snippet of behavioral profile - output of level-three rules for OneStepSearch 26

3.8 Converting text behavioural profiles into binary profiles. 26

3.9 Example of string type value (a), its corresponding hex representation (b), and binary representation (c) 27

4.1 List of process in a memory running Prolaco malware [75] 34

4.2 Snippet of IDT from memory running rustock botnet [75] 36

4.3 Snippet of GDT from memory running Alipop rootkit [75] 36

4.4 Device tree structure in a system running Stuxnet malware 37

4.5 Example of a malicious timer from a system running Prolaco malware. . . . 37

4.6 Snippet of SSDT in a system running BlackEnergy2 malware 40

4.7 Example of inline hook. 40

4.8 Examples of malicious callbacks in a system running BlackEnergy2 malware 42

5.1 UML diagram of the proposed malware classification framework. 45

6.1 Classification performance with regards to different parameters values. 55

6.2 Classification performance with regards to different parameters values 58

6.3 Number of rejected reports vs. different values of min_dist_cluster min_dist_cluster. 59

6.4 Classification performance excluding rejected samples. 60

6.5 Classification performance including rejected samples. 61

6.6 Number of rejected samples at the end of each day. 63

List of Symbols, Nomenclature or Abbreviations

max_dist_prototype	Maximum distance allowed between profiles and existing prototypes.
min_dist_cluster	Minimum distance required between a pair of clusters.
max_dist_classify	Maximum distance allowed between a profile and its closest prototype in order to be labeled.
reject_num	Minimum number of members expected in a cluster.
d(x, y)	Euclidean distance between profile x and y
$H(x_i, y_i)$	Hamming distance between value of feature i in the profile x and y.

Chapter 1

Introduction

1.1 Introduction

Malware, referred to any malicious software, poses a major threat to computer systems and the privacy of computer users. The damage caused by malware can differ from a minor increase in the outgoing traffic to a complete network breakdown and loss of critical data. Malware is extremely difficult to combat because it appears and spreads very quickly. Total number of unique malicious codes exceeds 28 million in the first quarter of 2015. According to McAfee Labs reports, the 2015 threat landscape is shaped by the sheer volume of new malware with sophisticated cyber espionage capabilities, including techniques of evading sandboxing technologies [39]. McAfee Labs detects more than 307 new threats every minute, and overall malware is increasing by 76 percent year after year. Given this rapid growth of malware, it is of paramount importance to quickly and accurately analyze unknown samples and understand their behaviour.

Malware analysis, as an important prerequisite for the development of automated detection tools, is the process of determining the purpose and primary functionalities of a given malware sample. Such functionalities include network activity, registry activity, and file activity,

just to name a few.

Malware analysis can be performed in two traditional ways: *static* or *dynamic* [62]. Static analysis is the process of examining a sample without running it to find basic information, e.g. strings, imported/exported libraries or functions. While straightforward and fast, malware can thwart static analysis by employing obfuscation techniques such as code packing, dead-code insertion, and code integration [79, 50].

In dynamic analysis, a sample is executed in a protected environment, e.g. sandbox, and its actual behavior is captured in the form of API/system calls, or in an instruction dump. Dynamic analysis of malware is immune to most obfuscation techniques and has shown to be more effective in differentiating malware families [30, 55, 56, 38].

Output of both static and dynamic analyses have been widely applied in the machine learning (ML) algorithms to group samples exhibiting similar behaviour [30, 55, 56, 38]. However, these approaches are suffer from the following problems:

1. Classification models using probabilistic or statistical characteristics of the training samples are unable to find similarity between two variants of the same family when there is a minor difference, e.g. presence of a dead code in one of the samples.

2. Effectiveness of these approaches depends on the employed analysis tools, e.g. malware that does not follow the traditional path of triggering API calls might not be visible to the sandbox. Escaping dynamic analysis, particularly evading sandboxes, has now become a significant analysis problem.

3. Proposed methods main objective is to profile malware behvaiour based on operations it performs. However, regardless of an action, impact of that operation on the environment is relatively important. Memory investigation, as an complementary analysis phase, can capture stealthy kernel level behvaiours that might not be observable by other types of analysis.

4. Output of these analysis tools is usually fed to ML algorithms directly, i.e. applying n-gram analysis or counting the frequency of calls. This ignores the fact that a single action can be performed in different ways resulting in execution of different API calls, e.g. a file may be accessed by its path or a handle to it.

5. Due to the voluminous data generated by each analysis tool, most of these approaches resort to only one analysis resource as the input to ML algorithms, limiting them from capturing other characteristics of a sample.

6. These approaches are based on low-level system events, e.g. individual system calls which are too low-level for abstracting semantically meaningful information and effectively describing the behaviour of malware.

In order to address these problems, we propose a framework that evaluates different aspects of a malware sample using both static and dynamic analysis techniques. Additionally, the proposed framework performs post-mortem analysis on the memory image acquired during malware execution to identify malicious kernel objects.

Outputs of these three types of analysis together provide a complete picture of an executable's activities, but require a cross-correlation step. Inconsistent and often redundant data in our framework are generalized and aggregated into a concise report, called the *behavioural profile*. The behavioural profile includes only required and interesting information about malicious code behaviour and can discriminate samples that exhibit similar behaviour more effectively. Then profiles are embedded in a high-dimensional vector space and are incrementally classified.

To demonstrate the applicability of the behavioural profiles, we compare our classification method with two other contemporary approaches using a reference dataset consisting of more than 13,000 unique executables from different malware families.

Due to the fact that Windows is considered as the most malware-ridden platform and at

the same time, most popular desktop operating system, our framework currently analyzes Windows executables, files conforming to the portable executable (PE) format. However, given the similarities in malicious activities of different types of malware, our framework can be extended to other types of operating systems.

1.2 Summary of Contributions

The main purpose of this thesis is to develop a new framework for intelligent malware classification. The contributions of this thesis can be summarized as follows:

- Defining high-level rules from the existing malware detection methodologies to effectively describe malware behaviour in a high-level form.
- Applying memory forensic analysis, as an additional information to the static and dynamic analysis.
- High-level interpretation of different analysis tools' output to discard less important aspects of malware behaviour, e.g. normal operations performed by all executables.
- Performing experimental evaluation on a large set of diverse malware samples. Our comparative analysis shows that:
    1. Aggregating the results of different analysis tools gives a more complete view of an executable's behaviour compared to only relying on one type of analysis.
    2. Differentiating items in the behavioural profile based on the discrimination power, significantly improves the classification performance in comparison to the simple text classification.

  The experimental results demonstrate the effectiveness of our classifier by an average classification accuracy of 81%, while leaving only 0.5% of samples as unlabeled.

1.3 Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 reviews machine learning-based malware classification approaches which employ static analysis, dynamic analysis, or a combination of both. Discussing some drawbacks of each group, behaviour-based classification approaches are introduced which focus on a specific malicious behaviour. These malicious behaviour can be grouped into eight categories: *replication and propagation, malicious arguments, persistancy, hooking, environment fingerprinting, process anomalies, registry access anomalies, and sensitive data access/modification.*

Chapter 3 provides a general overview of the framework, followed by a description of each module. It explains how an input binary sample is passed through three phases to be classified as an existing or a new malware family. First, different sources of information that can be extracted from an executable are explored. Then, our approach to cross-correlate output of different analysis tools is described and three levels of rules are discussed. Finally, the algorithm for incremental classification of received samples is explained.

Chapter 4 details the memory-based rules and different types of anomalies that can be detected from memory.

Design and implementation criteria of the proposed framework are described in Chapter 5. This Chapter introduces the main components and classes implemented in our framework and lists their main functions.

Chapter 6 reports the evaluation dataset, metrics, and comparative experimental results of the proposed framework with two existing malware classification approaches.

Finally, Chapter 7 concludes the thesis by discussing its contributions, limitations and possible improvements to the work done in this thesis.

Chapter 2

Literature Review

2.1 Static Malware Classification

With static analysis, the entire code can be covered and thus, possibly the complete behavior of a program can be captured, independent of any single path executed during run-time. Static analysis tools include disassemblers, decompilers, source code analyzers, and even basic utilities such as *strings* or *PEiD*[1]. Static information is extracted either directly from the binary (e.g. DLL imported or strings) [61] or after disassembly of the sample in the form of machine level instructions. To make the instruction dump compatible with ML algorithms three common approaches are: (1) extracting numeric features such as functions length, functions frequency [71] or opcode frequency [68]; (2) creating control/data flow graphs from the disassembled sample [36, 15, 11]; and (3) applying N-gram (a set of co-occuring words within a given window) on the instruction dump [61, 6, 38, 78, 80, 28].

The main weakness of static analysis is that the code analyzed may not necessarily be the code that is actually executed, e.g. when files are packed or a third party code is being downloaded and executed. Also, malware can employ a wide range of obfuscation mechanisms

[1]PEiD detects most common packers and compilers for PE files.

that make static analysis ineffective. Most recent researches have widely demonstrated the inefficiency of the static-based methods in the analysis of sophisticated malware [50, 46].

2.2 Dynamic Malware Classification

Due to the fundamental limitations of the static analysis [50, 46], most recent detection or classification approaches are based on dynamic execution of malware. The execution trace of an executable is normally captured in the form of machine level instructions, or API/system calls[2] [13]. Egele et al. [23] categorize different dynamic analysis techniques as: *Function call monitoring, Function parameter analysis, Information flow tracking, Instruction trace,* and *Autostart extensibility points.* Regardless of the technique employed, the effectiveness of dynamic analysis is influenced by several parameters [13].

- *Order of function calls.* Function calls can be considered either as a sequence, where order of calls occurrence is important [26], or as a group of short behaviour (in any order) [34]. Considering function calls in any order may increase the detection coverage, even though, it would result in a higher false positive rate [51].

- *Inclusion/exclusion of arguments.* Arguments of functions calls are included in the analysis when the relationship between function calls is important, e.g. deriving a dependency graph. But they can make the detection model too specific. Excluding arguments, in contrast, provides more generality, but can cause higher false positive rates accordingly [54]. Including only useful parts of arguments is another approach, which requires domain knowledge [56].

- *Biased vs unbiased analysis.* The majority of existing approaches take into account the presence or absence of function calls regardless of their importance [56, 26, 34]. Al-

[2]From now on we use "function calls" to refer to both API and system calls

though straightforward and independent from the domain knowledge, these approaches are more prone to the obfuscation techniques such as junk code injection. Only a small number of current studies discriminate function calls based on their criticality degree [37]. One is the host based anomaly detector engine proposed by Mutz et al. [51] that calculates the anomaly score of system calls based on the length of string arguments, distribution of the characters, and type of characters (printable or not).

- *Transparency.* Current dynamic analysis solutions such as ThreatAnalyzer [76], Aubis [10], Norman Sandbox [5], and Cuckoo [17], provide detailed results outlining what a sample does when executed inside an isolated environment. Providing a transparent environment to let malware reveal all of its intended activities is not trivial. User level analyzers for example, are only able to capture API calls invoked by malware and can be evaded by many kernel-level malware [23].

- *Analysis method.* Similar to the static instruction dump, dynamic traces can be processed by generating dependency graphs, passing n-grams, counting function call frequency, or matching them against predefined malicious patterns [56, 72, 63, 49, 18, 77].

Dynamic analysis compensates for most limitations of the static analysis, however, it is still inefficient in many cases, e.g. when arguments are encrypted, or malware performs its intended activities without calling ordinary API calls.

2.3 Hybrid Malware Classification

Assuming it is hard for malware to manipulate different types of features simultaneously, some approaches employ a combination of static and dynamic analysis. Such an approach has been proposed by Islam et al. [30] using function length frequency and printable strings, in combination with runtime API frequency.

Table 2.1: Summary of malware detection/classification approaches

Type of analysis	Method of feature extraction	Representation method	Examples
Static	Directly from PE	Numeric value	[61] [70]
	Disassembly of sample	Frequency	[71]
		N-gram	[6] [38] [78] [68] [61] [6] [38] [78] [80] [28]
		Graph	[36] [15] [11] [37] [21] [24] [12] [35] [14]
Dynamic	Execution of sample	Numeric value (frequency)	[72] [34] [51]
		N-gram	[26] [54] [56]
		Scenario based	[37] [77] [22] [49]
		Graph	[8]
Hybrid	Combination of above methods	Numeric value	[30]
		Graph	[60]

Table 2.1 summarizes malware analysis approaches with respect to the type of analysis, how features are extracted, and the way input data is prepared for the classification algorithm.

2.4 Behaviours-based Malware Detection

The output of analysis tools, e.g in the form of function calls or instructions, are still too low-level to describe what exactly an executable does on a system, hence, directly applying these results can make detectors vulnerable to many obfuscation mechanisms. Another group of studies falls into behaviour based malware detection, where semantically meaningful information is extracted from the low-level analysis results [9]. Self-replication, privacy invasion, or persistence are examples of malicious behaviour patterns emphasized in many real-time detection approaches.

These behaviour-based approaches are mostly process-oriented and only correlate the system calls or instructions invoked by a single process to determine if a process is malicious. In addition, they focus only on a few behvaiours (e.g. spawned processes, modified files, or

modified registry keys) and are able to detect limited categories of malware, e.g. viruses or rootkits [43, 49, 63, 64, 32].

Common malicious behaviour studied in the existing research and can be broadly grouped into: *replication and propagation, malicious arguments, persistancy, hooking and DLL injection, environment fingerprinting, process anomalies,* and *sensitive data access/modification.*

2.4.1 Replication and Propagation

Replication and propagation as two common symptoms of malware (commonly seen in viruses and worms) have been used in many detection approaches [43, 49, 63, 64]. Jacob et al. [31] grouped different types of self-replication methods into three categories:

1. Duplication: when malware copies itself to other places. Virus detection system proposed by Morales et al. [49] detects *duplication* by monitoring read and write attempts made by a process to copy itself to other places.

2. Infection : when malware injects itself into an existing file or process. Following three methods of replication, mentioned by Skormin et al. [63], can also be placed under the *infection* group:

   - Overwriting: when a virus overwrites itself into another existing file.
   - Companion : when a virus renames itself as another existing file.
   - Parasitic (locally): when a virus injects its code into another file and changes the entry point of the file.

3. Propagation: when malware replicates itself over the network, also called *over network parasitic* [63]. Approach proposed by Skormin et al. covers both *infection* and *propagation* [63] by defining possible scenarios of replication and checking them against a set of corresponding function calls.

2.4.2 Malicious Arguments

For simplicity reason or to decrease specificity, function calls are usually analyzed without their arguments which can lead to the failure in detection of attacks that carry a normal sequence of system calls but illegitimate arguments [51]. The approach proposed by Mutz et. al learns different aspects of arguments, e.g. length or distribution of characters, to detect malicious API or system calls [51]. Trinius et. al [74] suggest a flexible representation for API calls, called MIST, which allows some parts of arguments to be included in the analysis.

2.4.3 Persistancy

Modifying auto start extensibility points (ASEPs) is a common way for malware, especially spyware and adware, to make themselves persistent even after system reboot [43, 77]. Wu et al. define two cases of ASEP modification as 1) when malware directly makes itself an auto-start application, or 2) when it asks other popular auto-run applications to execute it [77]. Note, there are lots of benign applications with ASEP modification characteristics, but the main difference is that they do not hide, recover or reinstall themselves after being terminated by the user. There are several methods malware can employ for self-healing among which are:

- ASEP modification using registry keys: Changing auto-start registry keys is a prevalent method for applications to set ASEPs [77]. A list of popular auto-registry keys can be found in Appendix B.

  ASEP modification itself is not necessarily malicious, but there are some malicious activities associated with this modification that can make it suspicious. Excessive modification of the same auto-run key is an example of self-healing behaviour [77]. STARS Spyware detector, cleans any suspicious registry modification and checks if the process modifies that key again [77]. MAPMon [18] tags a process as malicious if there

exists an ASEP modification, very shortly after execution, associated with the process itself or its child processes. It only monitors API calls that might be used for registry key modification[3], and checks their behaviour against a prebuilt normal model. Some malware, however, are able to set ASEPs without calling Windows APIs.

- Copy the executable into the start-up folder: A program can also be persistent by copying itself into the start-up folders.

- Use executables to recover their ASEP entries [77]: To remain undetectable, malware may ask another process to change an ASEP on its behalf, or it may create a key then use its child processes to retrieve it.

- Paired self-healing processes: Malware sometimes creates a bunch of child processes, called *rescue team*, that recover each other by calling themselves continuously. This way if a process is stopped or removed another process calls it again. IBIS toolbar and eXact are examples of malware with paired execution behaviour [77].

2.4.4 Hooking and DLL injection

A program can import functions from other libraries statically, dynamically, or at run-time [62]. Malware usually engages in a covert way to hide injected DLLs. Possible methods of DLL injection in Windows can be summarized as follows:

- Using another process to load a DLL: Some Windows process such as Run32dll.exe and Svchost.exe can be used to load a DLL.

- Using process manipulation functions [18, 1]: Using CreateRemoteThread in combination with other functions such as LoadLibrary, a DLL can be injected into the address

[3]Examples of API calls used for registry key modification are: CloseHandle, CopyFile, CreateFile, CreateHardLink, MoveFile, MoveFileWithProgress, WriteFile, RegCloseKey, RegCreateKey, RegOpenKey, RegSetValue.

space of a process [4]. This method requires an executable to do the injection process [4].

- Using Windows hooking functions: Malware can request windows APIs such as SetWindowsHookEx to hook its malicious code on its behalf [42].

- Using debugging functions: Malware can call debugging functions, such as SuspendThread or NtSuspendThread, to suspend the threads of a process and modify their context using SetThreadContext or NtSetContextThread functions [59, 1].

- Preloading DLL injection: By not providing the full path of a DLL when calling functions such as LoadLibrary, Windows tries to search through some predefined directories including application's working directory. This makes it possible for malware to place a malicious DLL in one of those search directories [48].

- Modifying specific registry keys: DLLs listed under

  AppInit_DLLs key, for example, are loaded into all processes that load User32.dll.

- Modifying Import Address Table (IAT): IAT is a lookup table for an executable consisting of pointers to the required functions. This table can be modified to point to a malicious DLL.

- Application specific DLL injection: A DLL can be injected using popular applications such as browsers. Browser Helper Objects (BHO) plug-ins in IE, for example, are of interest to the attackers as they are loaded whenever IE starts.

- Attacking system call table: Levine et al. [41] summarize common methods of attacking system call table as follows: 1) *table modification*, when malware replaces a system call address with an address to its malicious system call, 2) *target modification*, when malicious code is being overwritten. The replaced code usually has a jump instruction

to the rest of malicious code, and 3) *table redirection*, when another fake system call table is created.

- Attacking Interrupt Descriptor Table (IDT): The Interrupt Descriptor Table (IDT) is a data structure in the memory, based on which processor transfers execution of a program to the respective interrupt handler when an interrupt or exception happens. Rootkits usually hook an IDT entry by changing a pointer in the table or modifying the handler code. Integrity of IDT is usually checked against a valid copy of IDT table [33, 3, 44], which is vulnerable to two issues:
  - Valid copy may be manipulated.
  - These approaches only check the integrity of pointers, i.e. if they point to an unusual part of the memory, but not the integrity of the handler code.

  IDTchecker, proposed by Ahmed et al. [7], is a rule-based IDT integrity checker tool designed for the cloud environment. It checks the integrity of both pointers and handler codes by comparing IDT status of all active VMs.

- Attacking System Service Dispatch Table (SSDT): SSDT table contains pointers to the kernel functions, and similar to IDT, may be modified to point to other parts of the memory.

- IRP function pointer modification: Windows drivers use structure called I/O request packets (IRP) to communicate with operating system and each other [19]. For each driver, there are some major functions that receive IRPs as input and tell the driver what to do. Table of pointers to the major functions can be hooked by replacing the address of a function [65].

- Hardware breakpoints: A thread can have a hardware breakpoint and be attached to a

debugger. In this case instead of modifying the original function's code, a breakpoint is set on the address of the hooked function.

2.4.5 Environment Fingerprinting

As recent sandboxes try to be more transparent, malware authors also employ more sophisticated fingerprinting techniques to detect a virtualized environment from the real machine. Human interaction is the most critical and hard to mimic type of fingerprinting, where malware checks the mouse speed, the number of mouse clicks, or waits until the user scrolls a document [2].

Although these types of fingerprinting are very hard to be detected from APIs, a few possible scenarios can be expected. For example, *GetAsyncKeyState* function is expected to be used to count the number of clicks, or WH_MOUSE_LL can be passed to the *SetWindowsHookExA* function to capture low level mouse inputs. *GetCursorPos* and *MessageBoxEx* function can also be invoked to check the mouse speed or existence of the message boxes, respectively [2]. Malware can also detect a virtualized environment by querying its name (using GetModuleFilename) and check if it is the same as the default string assigned to samples in the sandboxes. The serial number of the hard drive and unique services/files in sandboxes are other signs that malware can examine [2].

Malware sometimes prevents execution by exploiting specific characteristics of the virtualized environment, e.g. delaying execution to more than the default analysis time of a sandbox using functions such as Sleep, CreateTimerQueueTimer, or NtDelayExecution.

2.4.6 Process Anomalies

Running processes are interesting points to investigate in a compromised system. To avoid detection, some malware generate a process with the name similar to or exactly the same

as a known system process. Knowing OS core processes and their main characteristics, such as the expected number of instances, the default path, or the process's owner, are useful in detection of this type of malware. Zeus malware, for instance, is detectable by its svchost process that tries to connect to an external IP[4] [29]. Some characteristics and main functionalities of the Windows core processes have been summarized in Appendix C. Process hiding is another widely seen suspicious behaviour of many stealthy rootkits [32], which cannot easily be detected through API calls analysis.

2.4.7 Registry Access Anomalies

The registry in Windows contains important information, such as system and software configurations or security policies, that if is compromised, can affect a system's normal operation and cause harmful damages. Examples of suspicious registry modifications are as follow:

- Adding full read or write sharing access to the file system (by changing the networking section): This can be done through *HKEY_LOCAL_MACHINE/ Software/Microsoft/ Windows/CurrentVersion/Network/LanMan* key.
- Accessing rare keys: Some registry keys, e.g. *HKLM/Security/Providor*, are accessed rarely usually once at the time of the system installation.
- Reading or modifying keys created by other programs: For example an adversary can disable antivirus, or read another programs' user names/passwords.
- Retrieving sensitive information: SAM is an example of a sensitive file containing hashes of system passwords and can be found at *%SystemRoot%/system32/config/SAM*.
- Creating and accessing a lot of new registry keys: New software normally create a lot of new registry keys at installation time, but this can be suspicious if it is done in a

[4]Svchost is a system process responsible for loading start-up services with no expected Internet activities.

stealthy way, e.g. without any installation GUI or interface.

2.4.8 Sensitive Data Access or Modification

Similar to registry keys, access to or modification of sensitive data is a common malicious behaviour seen in many malware. Browser Helper Objects (BHOs) and toolbars, for example, are usually employed by spyware to get full control of the browser and steal important information [37].

2.5 Concluding Remarks

Malware is a pervasive problem in distributed computer and network systems. Identification of malware variants provides great benefit in early detection. Machine learning techniques, which are the predominant approach employed in most detection or classification systems, differentiate malware variants through the use of static or dynamic analysis. Static analysis can be ineffective if malware undergoes the obfuscation techniques to hide its real malicious behaviour. In contrast, dynamic analysis does a better job in presenting invariant characteristics or patterns of the malware by capturing dynamic traces during execution of a sample in an isolated environment. Regardless of analysis type, effectiveness of machine learning approaches are inherently affected by the statistical distribution of the training samples, which can easily be disturbed by employing obfuscation techniques.

To address mentioned shortcoming, instead of relying on the statistical attributes, some approaches have tried to interpret short patterns as a more general and higher level behaviour, which assumed to be consistent for a specific malware type or family. Concentrating on an absolute behaviour of a specific category, these approaches are not able to detect other types of malware. In this chapter, eight major groups of malicious activities have be discussed.

Chapter 3

Proposed Framework for Intelligent Malware Classification

3.1 Overview

Freely available analysis tools increase the capability of analysts to correlate different signs and symptoms of malicious behaviour. They save time and provide an overview of the sample's capabilities, so that analysts can decide where to focus their manual analysis efforts. However, output of these tools (usually in the text format) are not directly applicable for the classification process. Extracting statistical features, such as n-grams, is common method for transforming the output of these analysis tools into a numerical feature vector suitable for the similarity measure. These statistical features represent short behavioural patterns of malware but are too low-level to outline high-level behaviours such as self-replication, persistency, or process hiding.

In this chapter a new malware classification framework is proposed. The framework addresses some of the shortcomings of the existing classification methods by allowing output of various analysis tools to be correlated and interpreted using rules derived from existing domain

knowledge.

Figure 3.1 illustrates the general overview of the proposed intelligent malware classification framework which includes the following 3 phases:

- *Analysis*: suspicious binary is concurrently sent to three static, dynamic, and memory forensic analyzers. Outputs of these analyzers are then unified in a single JSON report.

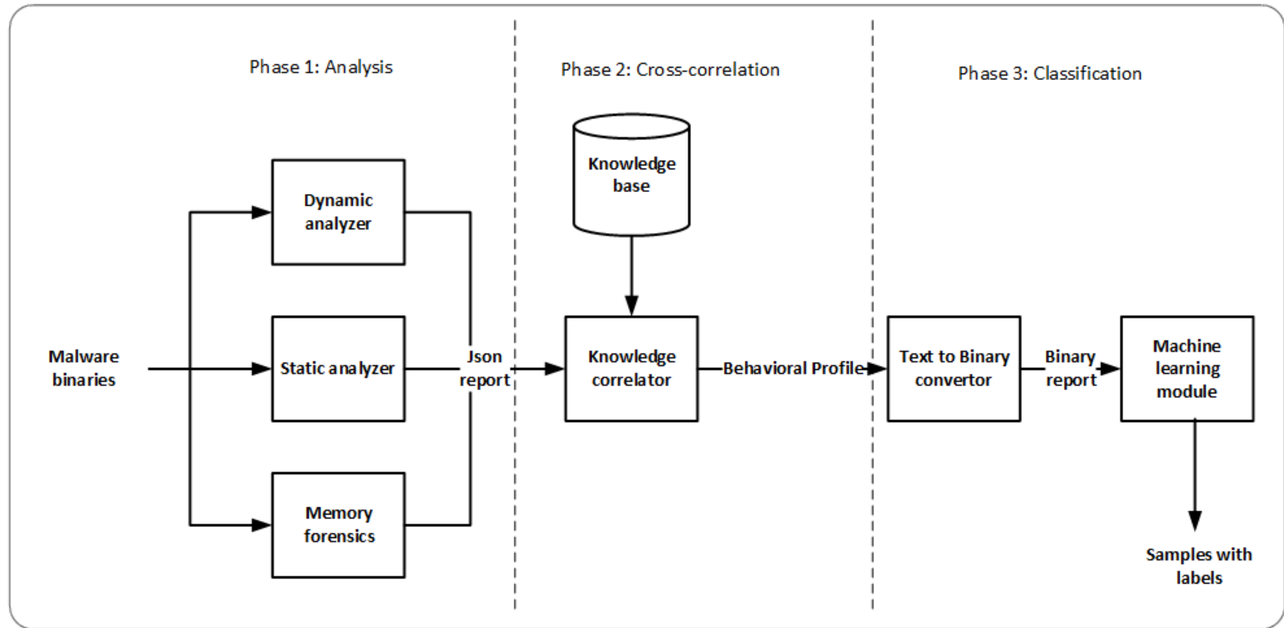

Figure 3.1: General overview of the proposed malware classification framework

- *Cross-correlation*: a number of rules are applied to the JSON analysis report to discover and abstract significant behaviours of a sample in a high-level report called *behavioral profile*.

- *Classification*: behaviour profile is transformed to a binary representation suitable for distance measurement and classified using ML algorithms.

3.2 Analysis

In the analysis phase, input binary is passed to the three complementary static, dynamic, and memory analyzers in parallel. The static analyzer examines the code and the structure of the executable to determine its functionality. For example, the entropy of header sections determines if the sample is packed or imported DLLs indicate the potential activities of the sample.

The dynamic analyzer provides insight that would be more difficult to obtain using only basic static analysis, e.g. actual actions performed, directory/files accessed, or URLs queried. The dynamic analyzer, however, is limited to the runtime functions and their arguments, which in many cases are not enough to state all events happening during execution. Performing deeper investigation on the user/kernel level memory, the memory analyzer is able to capture other artifacts such as threads' privilege, running processes, or current devices/drivers, which exclusively exist in the memory.

3.3 Knowledge Correlator

The knowledge correlator is the main module of the proposed framework. It is responsible for examining the JSON report generated during the analysis phase and correlating different indicators of a malicious behaviour. By utilizing the domain knowledge and available practical analysis or reverse engineering information we have defined three categories of rules. Level-one rules cover general and basic information that can directly be extracted from a binary, level-two rules highlight general, but not necessarily malicious behaviours. Finally, level-three rules are for deeper investigation to find stealthy and most likely malicious symptoms in an executable. Grouping rules in different levels gives us the flexibility of adjusting the different impact factor (weight) for each rule in the classification phase.

Rules are defined manually and can be extended by analysts once new information about

malware behaviour becomes available.

3.3.1 Level-one Rules

Figure 3.2 depicts an example of level-one rules output. Currently our level-one rules include:

- Process Tree Structure: shows the pairwise relationship between child processes created during execution. This rule is further used to detect self-healing behaviour.
- Sequence of static functions: lists important functions imported by the program (without executing the sample).
- Sequence of dynamic functions: Lists functions called during the execution of a sample.
- Size of analysis report: size of the report generated in the analysis phase.

```
    "Process Tree Structure": "(cd7a1609a341da70c8c8233311606022->GLB1.tmp)(GLB1.tmp-
>OSWDVA~1.EXE)(GLB1.tmp->DEALIO~1.EXE)(GLB1.tmp->VVSNInst.exe)(GLB1.tmp->RKINST~1.EXE)
(GLB1.tmp->iexplore.exe)",
    "Report size": "0.071884816"
    "Static MIST Sequence": "0d,0f,70,74,3e,2e,0c,29",
    "Dynamic MIST Sequence": "06 07 | 7c800000 0012ff20 00000000 00000000 00000000,06 02
| 00000000 07f96c13 7c800000 7c81cafa,06 02 | 00000000 0a39ecb1 7c800000 7c80b731,06 02
| 00000000 0b8e7db5 7c800000 7c809bd7,06 02 | 00000000 0803f0f3 7c800000 7c8309d1,06 02
| 00000000 072d1a44 7c800000 7c801ad4,06 07 | 7e410000 0012ff20 00000000 00000000
[snip]
```

Figure 3.2: A snippet of behavioral profile - output of level-one rules for OneStepSearch

The sequence of static or dynamic functions needs special treatment; instead of considering all functions, we only focus on the “interesting function calls”[1] from an analyst’s point of view and filter out common functions, which are seen in almost all executables. *CreateProcess* and *Sleep* are examples of interesting functions as they are commonly used by backdoors. On the other hand, imports from *msvcrt* DLL, which are included in almost every executable are eliminated. This filter provides two advantages in comparison to include all traces: 1) it thwarts obfuscation techniques such as dead code injection, and 2) it reduces the

[1] List of interesting functions can be found in Appendix A.

computational complexity of processing large reports. We also avoid text representation for static and dynamic traces and use MIST as an intermediate representation [74]. In MIST representation, different parts of an argument are arranged based on their stability [74]. Figure 3.3 shows the XML format of an example system call "*load_dll*". The corresponding MIST representation has been given in Figure 3.4, where from left to right specificity degree of attributes increases. As can be seen, stable attributes such as file extension and file path are placed in the higher (second) MIST tier, and attributes such as file name or file size are placed in the lower (third) MIST tier because they are less consistent and tend to vary in different variants of the same family. First MIST tier indicates the category and name of the system call. Category assigned to an API call may vary in different analyzers, but order of parameters stay the same in different implementation of MIST. Table 3.1 shows different API categories detected by CWSandbox and Cuckoo malware analyzers.

```
<load_dll filename="C:\WINDOWS\system32\kernel32.dll" successful="1"
 address="#7C800000" end_address="#7C908000" size="1081344"
 filename_hash="c88d57cc99f75cd928b47b6e444231f26670138f"/>
```

Figure 3.3: Example of system call generated by CWSandbox [56]

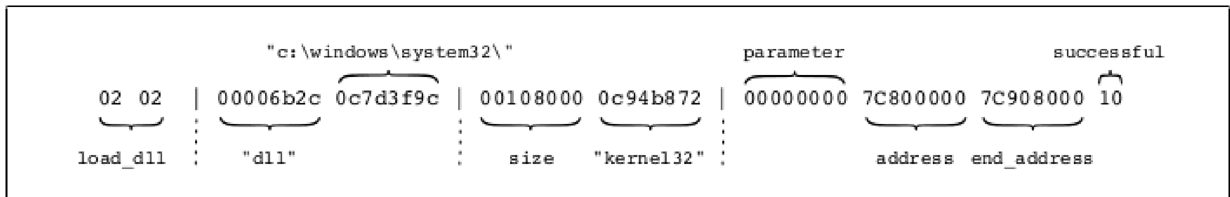

Figure 3.4: Example of MIST representation for system call *load_dll* [56]

MIST representation is defined as an XML file containing hex values corresponding to the category and name of API calls. Figure 3.5 shows MIST information for loadDll function in the "system" category. In the MIST representation, type of each parameter dictates in which tier of MIST representation it should be places. For instance, a parameter with type

Table 3.1: CWSandbox and Cuckoo comparison

CWSandbox		**Cuckoo Sandbox**	
Category	**No of System Calls**	**Equivalent Category**	**No of system calls**
Windows COM	4	-	-
DLL Handling	3	-	-
Filesystem	14	Filesystem	7
ICMP	1	-	-
Inifile	5	-	-
Internet Helper	5	-	-
Mutex	2	-	-
Network	6	Network	5
Registry	9	Registry	18
Process	7	Process	10
Windows Services	11	Services	9
System	2	System	8
System info	7	-	-
Thread	3	Threading	8
User	8	-	-
Virtual Memory	5	Virtual memory	3
Window	5	Windows	4
Winsock	13	-	-
Protected Storage	9	-	-
Windows Hooks	1	Hooking	2
-	-	Synchronization	2
-	-	Device	1
-	-	Socket	1

"path" is split in order to maintain file type and directory path in the tier two and shift the file name to tier three.

```
<system mist="02">

   [snip]

   <loadDll mist="02" weight="1">
        <baseaddress type="type_hex" />
        <flags type="type_integer" />
        <filename type="type_path" />
    </loadDll>

    [snip]
```

Figure 3.5: Example of MIST definition for system call *loadDll*

This granular arrangement of arguments provides several advantages:

- The granularity level of the analysis is adjustable up to a certain level, namely from coarse grained analysis, which considers system call with no arguments, to fine grained analysis that considers all arguments.
- The different parts of an argument can be weighted according to their degree of stability.
- Noisy arguments are removed from the analysis.

3.3.2 Level-two Rules

Figure 3.6 shows a snippet of output generated by level-two rules for a packed sample that searches through the file system[2], has network activity, and modifies 5 auto-run registry keys. To increase the certainty, this group of rules check all possible indicators of the same behaviour that may be reported by different analyzers. For example, packed executables can be detected by either checking their header against the predefined packers signature, or

[2]The value of 1 in the report means that sample has the corresponding behvaiour.

checking the naming convention of the header sections which should be consistent across the same compiler. Also a big difference between the *Virtual Size* and *Size of Raw data* attributes of each section is another characteristic of packed executables [62]. Another example is having GUI activities which can be observed from either GUI related API calls captured during dynamic analysis or imports from GDI32.dll reported by the static analyzer.


```
    "GUI manipulation": "0",
    "Searching Directories": "1",
    "SelfRuning": "0",
    "low-level keyboard event as the call back function": "0",
    "(potential) Getting permission": "0",
    "Network functionalities": "1",
    "(potential) Using registry": "0",
    "Executing programs": "1",
    "Loading Dll-debugging api": "0",
    "(potential) GUIactivity": "0",
    "registering hotkeys": "0",
    "(potential) Executing programs": "0",
    "packed": "1 (section anomaly and packer: Wise Installer Stub)",
    "Creating backdoor": "0",
    "Loading from resource": "0",
    "Opening/Manipulating process": "0",
    "Dynamic Linking": "0",
    "ASEPModification-AutoRun-setting reg AutorunKeys": "software\\microsoft\\windows\
\currentversion\\runonce,software\\classes\\clsid,software\\microsoft\\windows\
\currentversion\\explorer\\shellexecutehooks,system\\currentcontrolset\\services,software
\\microsoft\\windows\\currentversion\\run",
    "Opening/Manipulating file": "1",
    "Keylogging activity or Loading Dll-hooking api": "0",
    "(potential) Network functionalities": "0"
 [snip]
```


Figure 3.6: A snippet of behavioral profile - output of level-two rules for OneStepSearch

3.3.3 Level-three Rules

This group of rules concentrates on the most discriminative and most likely malicious characteristics of an executable such as hooking, malicious registry access, and process hiding. Figure 3.7 shows a part of level-three rules' output generated from OneStepSearch malware, which has two excessive auto-run key modifications, adds a number of new devices to an existing driver, and hooks IDT. The majority of level-three rules are extracted from the memory, which is the hardest place for malware to hide its footprints from. Memory-based rules are discussed in more detail in Chapter 4.

```
    "Self ReadWriteBased-DirectWrite": "0",
    "process anomaly-lsass.exe": "0",
    "Threads (Orphan)": "",
    "New device added to an existing driver": "(\\Driver\\hidusb->00000046-> - \\Driver\\mouhid),(\\Driver\\hidusb->00000046-> - \\Driver\
\VBoxMouse),(\\Driver\\hidusb->00000046->PointerClass2 - \\Driver\\Mouclass),(\\Driver\\swenum->KSENUM#0000000a-> - \\Driver\\kmixer),(\\Driver\
\ACPI->00000038-> - \\Driver\\isapnp),(\\Driver\\PCI->NTPNP_PCI0008->{C3DE9AA3-FCD4-4655-9CE1-D3EB8F12F3E7} - \\Driver\\PCnet),(\\Driver\\ACPI_HAL-
>00000034-> - \\Driver\\ACPI),(\\Driver\\PnpManager->00000032-> - \\Driver\\mssmbios)",
    "Paired Processes": "0",
    "Hidden dlls": "",
    "IDT Hooks TypeB": "(18->ntoskrnl.exe->.text),(32->unknown->),(33->unknown->),(34->unknown->),(35->unknown->),(36->unknown->),(37->unknown->),
(38->unknown->),(39->unknown->),(40->unknown->),(41->unknown->),(48->hal.dll->.text),(49->unknown->),(53->unknown->),(56->hal.dll->.text),(57-
>unknown->),(58->unknown->),(59->unknown->),(60->unknown->),(62->unknown->),(63->unknown->)",
    "IDT Hooks TypeC": "",
    "IDT Hooks TypeA": "",
    "Running rundll32": "0",
    "Dlls with abnormal path": "",
    "Frequency of Access/Modifictaion on ASEP": "software\\microsoft\\windows\\currentversion\\run|3 software\\classes\\clsid|31 ",
[snip]
```


Figure 3.7: A snippet of behavioral profile - output of level-three rules for OneStepSearch

3.4 Text to Binary Converter

Behavioral profiles generated by the knowledge correlator are in text format and not yet suitable for ML algorithms. These profiles are transformed into a hex and then binary profiles as depicted in Figure 3.8.

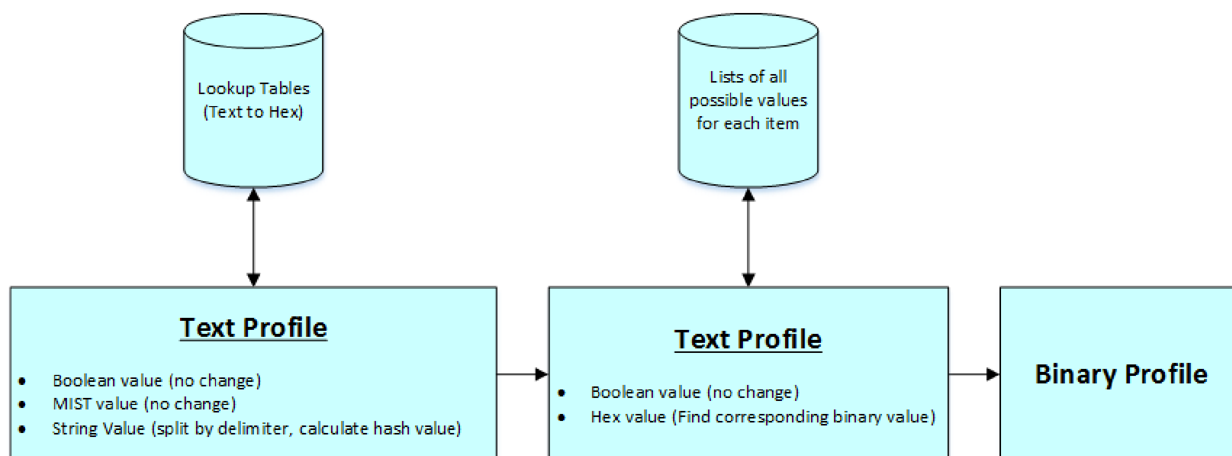

Figure 3.8: Converting text behavioural profiles into binary profiles.

A text profiles includes there types of value: boolean, string, and MIST. Boolean values do

not require any modification. String values are split by the corresponding delimiter (e.g. ‘,’ or ‘()’), and hash values of each parts is then replaced. Boolean representation of a string can then be obtained from a lookup table containing all possible values. Figure 3.9 shows an example of string type value, its corresponding hex, and boolean representations.

MIST values are in hex format so they are directly split by comma and each part is replaced by the corresponding boolean value.

```
(a)

    "Process with enabled privilege": "(winlogon.exe->SeSecurityPrivilege),
(winlogon.exe->SeLoadDriverPrivilege),(winlogon.exe->SeUndockPrivilege),
(lsass.exe->SeCreateTokenPrivilege),(lsass.exe->SeLoadDriverPrivilege),
(lsass.exe->SeUndockPrivilege),(VBoxService.exe->SeSystemtimePrivilege),
(svchost.exe->SeLoadDriverPrivilege),(svchost.exe->SeSystemtimePrivilege),
(svchost.exe->SeUndockPrivilege),(explorer.exe->SeLoadDriverPrivilege),
(explorer.exe->SeUndockPrivilege),(spoolsv.exe->SeLoadDriverPrivilege),
(spoolsv.exe->SeUndockPrivilege),(VBoxTray.exe->SeLoadDriverPrivilege),
(VBoxTray.exe->SeUndockPrivilege),(wcwbhhh.exe->SeLoadDriverPrivilege),
(wcwbhhh.exe->SeUndockPrivilege),(python.exe->SeLoadDriverPrivilege),
(python.exe->SeUndockPrivilege),(python.exe->SeDebugPrivilege),(python.exe-
>SeLoadDriverPrivilege),(python.exe->SeUndockPrivilege)",


(b)

"Process with enabled privilege": "0f4cbc69 00a5fab9 0de3f7f9 0d1f4f99 060c5ac9
04438059 0d2ca7f9 01396c39 001c7b19 017974a9 0768dae9 00c3e2b9 0237d9c9
0fc681a9 0427c619 0fdf1279 097c0029 041d2ce9 0cf42709 0c31cb69 0f29ba59
0cf42709 0c31cb69",

(c)

"Process with enabled privilege":
"1111111111111111111110000000000000000000000000",
```

Figure 3.9: Example of string type value (a), its corresponding hex representation (b), and binary representation (c)

3.5 Classification

The main objective of the ML module is to discover whether a newly acquired binary sample is a representative of a known family of malware, or whether it represents a new discovery. Our approach for classifying samples falls within the machine learning paradigm. We have considered the following criteria for the selection of our machine learning algorithm:

- Complexity of the model: With the tremendous number of suspicious samples received daily, and computational complexity involved in extracting features, selection of an algorithm with minimum complexity is required.

- Compatibility with the weighted features: ML algorithm needs to be compatible with weighted features, e.g. K-Nearest Neighbour and weighted SVM.

- Adaptability to the stream learning: ML algorithm is required to be compatible for the stream of data, as new samples are received continuously.

To accelerate learning, we follow the approximate classification algorithm proposed by Rieck et al. [56], referred to as *nearest prototype classification*, which resembles the costly K-Nearest Neighbor algorithm. In this algorithm, instead of considering all instances, representative instances, called prototypes, are extracted and used in the clustering and classification processes. Remaining instances are then labeled as their closest prototype.

3.5.0.1 Prototype Extraction

A prototype in our system is a behavioual profile that can represent its surrounding profiles. Extracting an optimal set of prototypes is NP-hard [56] and is usually performed by employing clustering algorithms. We use the linear-time prototype extraction algorithm suggested by Gonzalez [27] (Algorithm 1), where $distance[x]$ determines the distance between profile x and its nearest prototype. The algorithm starts with adding first profile in the training

set into the list of prototypes. Subsequently, farthest profiles are selected as prototypes one at a time, and *distance*$[x]$ is recalculated for each profile x. This process continues until the distance of all profiles from their closest prototype is less than the specified threshold d_p. The algorithm's run-time linearly increases by the number of profiles and prototypes.

Algorithm 1 Prototype extraction

```
prototypes ← ∅
distance[x] ← ∞ for all x ∈ profiles
while max (distance) ≥ max_dist_prototype  do
      choose z such that distance[z] = max(distance)
   for  x ∈ profiles and x ≠ z do
      if  distance[x] > d(x, z) then
         distance[x] ← d(x, z)
      end if
   end for
   add z to prototypes
end while
```

3.5.0.2 Clustering of Prototypes

In the clustering phase, only extracted prototypes are clustered to group similar malware families and identify unknown samples. Algorithm 2 describes the employed *hierarchical clustering* algorithm where prototypes with the distance less than the threshold d_c are merged into one cluster, then clusters with fewer than m members are rejected and held back for further analysis.

3.5.0.3 Classification

Similar malware families are grouped during the clustering phase to identify unknown samples; but to assign a clear description or label (i.e. family name) to each sample, we proceed to the approximate classification approach depicted in Algorithm 3, where classification is

[3]Distance between clusters are calculated based on the maximum distance between their individual members (prototypes).

Algorithm 2 Clustering Using Prototypes

```
for z, z′ ∈ prototypes do
    distance[z, z′] ← d(z, z′)
end for
while min(distance) ≤ min_dist_cluster do
    merge clusters z, z′ with minimum distance[z, z′]
    update distance using complete linkage³
end while
for x∈ profiles do
    z ← nearest prototypes to x
    assign x to cluster containing z
end for
reject clusters with less than m members
```

simply done by propagating each prototype's label to its corresponding members. Members (profiles), whose distance from the closest prototype is more than a predefined threshold d_r are rejected and kept for later incremental analysis. Similar to the prototype extraction, the algorithm's run-time linearly increases by the number of profiles and prototypes.

Algorithm 3 Prototype classification

```
for x ∈ profiles do
    z ← nearest prototype to x
    if d(z, x) ≥ max_dist_classify then
        reject x as unknown class
    else
        assign x to cluster containing z
    end if
end for
```

Considering the fact that malware are received as a stream, we process the incoming reports in small chunks, e.g., on a daily basis. To realize an incremental analysis, we need to keep track of intermediate results, such as prototypes extracted during previous runs of the algorithm. Algorithm 4 sketches the incremental classification procedure which starts by checking input reports against previous prototypes, then re-clustering the remaining reports.

Algorithm 4 Incremental classification

rejected ← ∅, prototypes ← ∅
for profiles ← new profiles ∪ rejected profiles **do**
  classify profiles to known clusters using prototypes
  extract prototypes from remaining profiles
  cluster remaining profiles using prototypes
  prototypes ← prototypes ∪ prototypes of new clusters
  rejected ← rejected profiles from clustering
end for

3.5.0.4 Profiles Distance

The distance between profiles x and y is calculated using *Euclidean* distance measure:

$$d(x, y) = \sqrt{w_1(x_1 - y_1)^2 + w_2(x_2 - y_2)^2 + ... + w_n(x_n - y_n)^2}$$

where n is the number of features (high level rules in the behavioural profile), x and y are two behavioural profiles, and w_i is the corresponding weight assigned to feature i. Having different types of features with different magnitude requires a normalization procedure to map all values to a unified scale (in our case a value between 0 and 1). For each binary sequence feature i with maximum length of l_i, distance between x_i and y_i is:

$$(x_i - y_j) = \frac{H(x_i, y_i)}{l_i}$$

where $H(x_i, y_i)$ is the *Hamming*[4] distance between value of feature i in the profiles x and y.

[4] In Hamming distance, number of positions where two strings have different value is counted.

3.6 Concluding Remarks

In this chapter we proposed a malware classification framework that enhances the traditional classification approaches by utilizing high-level domain knowledge. The proposed framework allows integration of different analysis tools in a form of concise yet meaningful behavioural profiles applicable in machine learning algorithms. In general, the proposed framework has the following advantages:

- Supports timely analysis and classification of streaming samples.
- Provides a comprehensive and high level overview of executables behaviour, saving considerable time and effort required in the manual malware dissection. These comprehensive reports can further be visualized to make pairwise comparison of samples easier.
- Improves the existing ML based detection or classification systems by utilizing domain knowledge.
- Allows future extension by adding other analysis tools as plug-ins and fully automating the analysis process.
- Makes rules extension possible when more information about malware becomes available.

Chapter 4

Memory Forensics

4.1 Overview

Most types of hookings (e.g. DKOM and hardware based approaches [40]), threads' privilege, or other artifacts such as executed commands, running processes, and current devices/drivers exist exclusively in memory. This chapter describes some types of anomalies that can be detected by applying memory-based rules on the memory image of the system running the sample.

4.2 Process Anomalies

Running processes can be retrieved from different memory resources such as $PsActiveProcessHead$ linked list or the pool tag. To remain stealthy, malware can hide its processes from one or some of these sources [75]. However, by comparing all sources of process listing, hidden or terminated processes can be detected. Figure 4.1 shows suspicious process *1_doc_RCData_61* that has unlinked itself from *pslist* but is present in all other sources.

We also check hosting DLL and full path of services/processes, against a list of known facts

Offset(P)	Name	PID	pslist	psscan	thrdproc	pspcid	csrss	session	deskthrd
0x06499b80	svchost.exe	1148	True	True	True	True	True	True	True
0x04b5a980	VMwareUser.exe	452	True	True	True	True	True	True	True
0x05f027e0	alg.exe	216	True	True	True	True	True	True	True
0x010f7588	wuauclt.exe	468	True	True	True	True	True	True	True
0x04c2b310	wscntfy.exe	888	True	True	True	True	True	True	True
0x061ef558	svchost.exe	1088	True	True	True	True	True	True	True
0x06015020	services.exe	676	True	True	True	True	True	True	True
0x06384230	vmacthlp.exe	844	True	True	True	True	True	True	True
0x069d5b28	vmtoolsd.exe	1668	True	True	True	True	True	True	True
0x04a544b0	ImmunityDebugge	1136	True	True	True	True	True	True	True
0x0655fc88	VMUpgradeHelper	1788	True	True	True	True	True	True	True
0x06945da0	spoolsv.exe	1432	True	True	True	True	True	True	True
0x05f47020	lsass.exe	688	True	True	True	True	True	True	True
0x0113f648	1_doc_RCData_61	1336	False	True	True	True	True	True	True

Figure 4.1: List of process in a memory running Prolaco malware [75]

about Windows core processes. This is useful for cases in which a malicious process with the same name as a legitimate process is created [67], or when malware installs a service by calling a legitimate process and then poisons it by malicious contents [75].

4.3 DLL Anomalies

Currently we detect two dominant types of DLL anomalies:

- DLL hiding: A reference to a loaded DLL is kept in one or all of 3 lists of the Process Environment Block (PEB): InLoadOrderModuleList, InInitOrderModuleList, and InMemoryModuleList. Since PEB is in user level, malware can easily remove its DLL from these lists. However, the base address of the DLL can still be retrieved from VAD information helping in detection of hidden DLLs.

- Abnormal DLL path: Instead of unlinking a DLL, malware can overwrite known DLLs. By comparing the full path of the DLL to what is reported by each list, any inconsistency can be detected.

4.4 Attacking IDT/GDT

Rootkits usually hook an IDT entry by either changing the pointer or modifying the handler code. A valid copy of this table is made before running the sample. Then following rules are applied to detect abnormal IDT behaviours:

- Handler code should reside in *.text* section of the module. Hence, entries having a handler in other sections, e.g. entry with *.rsrc* section in Figure 4.2, are extracted.
- An interrupt handler code should exist within the basic kernel code address space or kernel drivers/modules [7]. To find the handler's base address for each entry, we use *Segment Selector* value which is an index to the corresponding handler in the Global Descriptor Table (GDT) table. GDT can also be directly compromised, e.g. by installing a call gate and allowing user mode programs to run in the kernel mode [75]. Therefore, GDT entries, such as *CallGate32* shown in Figure 4.3, with type other than *TSS, Code, Data, or Reserved* are extracted.
- Based on our observations, there are a limited number of distinct owning module names (ntoskrnl.exe, hal.dll, or Unkown). Entries with other module names are selected.
- The first 32 interrupt handlers are identical in the same kernel, hence they can be checked against the clean image of the analyzer's kernel. Other entry points (32-255) are user specified and are not expected to have the same code [7].

4.5 New Drivers/Devices

Drivers and devices are interesting points for rootkits to exploit. They can add new drivers or add new devices to an existing driver for many reasons such as hiding network connections or capturing mouse/keyboard movements [75]. By using the image of all drivers and devices

CPU	Index	Selector	Value	Module	Section
0	0	8	0x8053e1cc	ntoskrnl.exe	.text
0	1	8	0x8053e344	ntoskrnl.exe	.text
0	2	88	0x00000000	ntoskrnl.exe	
0	3	8	0x8053e714	ntoskrnl.exe	.text
0	4	8	0x8053e894	ntoskrnl.exe	.text
0	5	8	0x8053e9f0	ntoskrnl.exe	.text
0	6	8	0x8053eb64	ntoskrnl.exe	.text
0	7	8	0x8053f1cc	ntoskrnl.exe	.text
0	8	80	0x00000000	ntoskrnl.exe	
[snip]					
0	2B	8	0x8053db10	ntoskrnl.exe	.text
0	2C	8	0x8053dcb0	ntoskrnl.exe	.text
0	2D	8	0x8053e5f0	ntoskrnl.exe	.text
0	2E	8	0x806b01b8	ntoskrnl.exe	.rsrc

Figure 4.2: Snippet of IDT from memory running rustock botnet [75]

CPU	Sel	Base	Limit	Type	DPL	Gr	Pr
0	0x0	0x00ffdf0a	0xdbbb	TSS16 Busy	2	By	P
0	0x8	0x00000000	0xffffffff	Code RE Ac	0	Pg	P
0	0x10	0x00000000	0xffffffff	Data RW Ac	0	Pg	P
0	0x18	0x00000000	0xffffffff	Code RE Ac	3	Pg	P
0	0x20	0x00000000	0xffffffff	Data RW Ac	3	Pg	P
0	0x28	0x80042000	0x20ab	TSS32 Busy	0	By	P
0	0x30	0xffdff000	0x1fff	Data RW Ac	0	Pg	P
0	0x38	0x00000000	0xfff	Data RW Ac	3	By	P
0	0x40	0x00000400	0xffff	Data RW	3	By	P
0	0x48	0x00000000	0x0	<Reserved>	0	By	Np
[snip]							
0	0x3d0	0x00008003	0xf3d8	<Reserved>	0	By	Np
0	0x3d8	0x00008003	0xf3e0	<Reserved>	0	By	Np
0	0x3e0	0x8003f000	0x0	CallGate32	3	-	P
0	0x3e8	0x00000000	0xffffffff	Code RE Ac	0	Pg	P
0	0x3f0	0x00008003	0xf3f8	<Reserved>	0	By	Np
0	0x3f8	0x00000000	0x0	<Reserved>	0	By	Np

Figure 4.3: Snippet of GDT from memory running Alipop rootkit [75]

before infection, new objects can be detected. Figure 4.4 shows an example of an unnamed device, with its three attached devices, added by Stuxnet to $nFileSystemnNtfs$ driver.

```
DRV 0x0253cee8 \Driver\Ftdisk
---| DEV 0x82256900 HarddiskVolume1 FILE_DEVICE_DISK
------| ATT 0x81de19b8  - \Driver\VolSnap FILE_DEVICE_DISK
---| DEV 0x820cd788 FtControl FILE_DEVICE_NETWORK
DRV 0x0253d180 \FileSystem\Ntfs
---| DEV 0x82166020  FILE_DEVICE_DISK_FILE_SYSTEM
------| ATT 0x8228c6b0  - \FileSystem\sr FILE_DEVICE_DISK_FILE_SYSTEM
---------| ATT 0x81f47020  - \FileSystem\FltMgr FILE_DEVICE_DISK_FILE_SYSTEM
------------| ATT 0x81fb9680  - \Driver\MRxNet FILE_DEVICE_DISK_FILE_SYSTEM
---| DEV 0x8224f790 Ntfs FILE_DEVICE_DISK_FILE_SYSTEM
------| ATT 0x81eecdd0  - \FileSystem\sr FILE_DEVICE_DISK_FILE_SYSTEM
---------| ATT 0x81e859c8  - \FileSystem\FltMgr FILE_DEVICE_DISK_FILE_SYSTEM
------------| ATT 0x81f0ab90  - \Driver\MRxNet FILE_DEVICE_DISK_FILE_SYSTEM
DRV 0x02599f38 \Driver\IntelIde
---| DEV 0x820cde60 PciIde0Channel1-1 FILE_DEVICE_CONTROLLER
------| ATT 0x82156030 IdePort1 - \Driver\atapi FILE_DEVICE_CONTROLLER
---| DEV 0x820cd030 PciIde0Channel0-0 FILE_DEVICE_CONTROLLER
------| ATT 0x822dd030 IdePort0 - \Driver\atapi FILE_DEVICE_CONTROLLER
---| DEV 0x8229e2d0 PciIde0 FILE_DEVICE_BUS_EXTENDER
```

Figure 4.4: Device tree structure in a system running Stuxnet malware

4.6 Anomalous Timers

Another stealthy method of hooking is to register a kernel timer with a nefarious procedure. We expect the module name of a timer to end with .sys or .exe, and also the procedure of the timer should be within the kernel space, otherwise it is malicious. Figure 4.5 shows a malicious timer whose procedure is not in a valid range and has an *Unknown* module name.

```
Offset(V)  DueTime                  Period(ms) Signaled   Routine    Module
---------- ------------------------ ---------- ---------- ---------- ------
0xffafb188 0x00000063:0x0e3cc168             0 -          0xfc2b592e sr.sys
0x80552408 0x80552418:0x80552418    -214...832 Yes        0xff12f128 UNKNOWN
0x8054f288 0x00000063:0x0e43e876             0 -          0x804e5aec ntoskrnl.exe
0x8055a300 0x00000064:0x956bb616             0 -          0x80533bf8 ntoskrnl.exe
```

Figure 4.5: Example of a malicious timer from a system running Prolaco malware.

4.7 Hidden drivers

A list of system drivers can be extracted using either the linked list pointed to PsLoaded-ModuleList, which fails to find hidden or unlinked drivers, or pool tag scanning that is able to find all drivers. By comparing the drivers reported by these two sources, unlinked or unloaded drivers can be extracted.

4.8 Thread Anomalies

Extensive details on a system's threads, including the value of each register or thread's fields can be captured from the memory dump. Any given system has hundreds of threads which make it difficult to sort through. Thread related rules in our system only focus on the threads with abnormal or important characteristics as follows [45]:

- Orphan threads: Threads that unlink their owning kernel driver.
- Threads with hooked SSDT: The ServiceTable field of a thread points to an array of virtual addresses called SSDT where each entry points to a kernel function. One common way to evaluate if this table is intact is to check the range of the address for each entry and make sure it is inside the kernel address space.
- Hidden threads (using DKOM technique): It is common for rootkits to unlink themselves from OS's list of threads by using DKOM technique, i.e. modifying a thread's ExitTime. By checking other fields of a thread such as _KTHREAD_STATE, it can be verified if a thread actually exited.
- Hidden thread (from debugger): Malicious threads usually hide themselves from the debugger to prevent reverse engineering.

- Threads with hardware breakpoints: A thread may set a hardware breakpoint and be attached to a debugger. In this case instead of modifying the original function's code, a breakpoint is set on the address of the hooked function.

4.9 SSDT Anomalies

Currently, the following rules are applied to detect SSDT anomalies:

- Owning module: Windows has 4 SSDTs but only 2 of them are used, one by *ntoskrnl.exe* and the other by *win32k.sys* module.
- Table duplication: There should not be more than one SSDT table with the same index. This can happen if malware tries to create another shadow table. Figure 4.6 shows how BlackEnergy2 attacks SSDT by duplicating SSDT at index 0.
- Number of tables and entries: Tables should have a fixed number of entries in the same OS version.

4.10 API Hooks

Different types of hooking, including IAT, EAT, and Inline style hooks, can be detected from CALLs and JMPs instructions to direct and indirect locations in memory or the module associated to the hook. Figure 4.7 shows an example of Inline hook. The hooking module is unknown because there is no module (DLL) associated with the memory in which the rootkit code exists.

```
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at ff3aab90 with 284 entries
  Entry 0x0000: 0x8059849a (NtAcceptConnectPort) owned by ntoskrnl.exe
  Entry 0x0001: 0x805e5666 (NtAccessCheck) owned by ntoskrnl.exe
  Entry 0x0002: 0x805e8ec4 (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe
  Entry 0x0003: 0x805e5698 (NtAccessCheckByType) owned by ntoskrnl.exe
[snip]

SSDT[0] at 80f162d0 with 284 entries
  Entry 0x0000: 0x8059849a (NtAcceptConnectPort) owned by ntoskrnl.exe
  Entry 0x0001: 0x805e5666 (NtAccessCheck) owned by ntoskrnl.exe
  Entry 0x0002: 0x805e8ec4 (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe
  Entry 0x0003: 0x805e5698 (NtAccessCheckByType) owned by ntoskrnl.exe
[snip]

SSDT[0] at 80501030 with 284 entries
  Entry 0x0000: 0x8059849a (NtAcceptConnectPort) owned by ntoskrnl.exe
  Entry 0x0001: 0x805e5666 (NtAccessCheck) owned by ntoskrnl.exe
  Entry 0x0002: 0x805e8ec4 (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe
  Entry 0x0003: 0x805e5698 (NtAccessCheckByType) owned by ntoskrnl.exe
  Entry 0x0004: 0x805e8efe (NtAccessCheckByTypeAndAuditAlarm) owned by ntoskrnl.exe
[snip]
SSDT[1] at bf997600 with 667 entries
  Entry 0x1000: 0xbf934ffe (NtGdiAbortDoc) owned by win32k.sys
  Entry 0x1001: 0xbf946a92 (NtGdiAbortPath) owned by win32k.sys
  Entry 0x1002: 0xbf8bf295 (NtGdiAddFontResourceW) owned by win32k.sys
  Entry 0x1003: 0xbf93e718 (NtGdiAddRemoteFontToDC) owned by win32k.sys
  Entry 0x1004: 0xbf9480a9 (NtGdiAddFontMemResourceEx) owned by win32k.sys
[snip]
```

Figure 4.6: Snippet of SSDT in a system running BlackEnergy2 malware

```
Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 1884 (IEXPLORE.EXE)
Victim module: WININET.dll (0x771b0000 - 0x77256000)
Function: WININET.dll!HttpAddRequestHeadersA at 0x771c54ca
Hook address: 0xf90000
Hooking module: <unknown>
[snip]
```

Figure 4.7: Example of inline hook.

4.11 Malicious Callbacks

Similar to some kernel components or anti-viruses, malware can use kernel callbacks such as PsSetCreateProcessNotifyRoutine, PsSetCreateThreadNotifyRoutine, or IoRegisterShutdownNotification to receive a notification when a specific event (e.g. process or thread creation or shutdown) happens and react correspondingly [75]. The owning module of a callback is an important parameter determining malicious callbacks. Figure 4.8 shows the *callback* list of a system infected by BlackEnregy2 malware. Two malicious callbacks PsSetCreateThreadNotifyRoutine at 0xff0d2ea7 and GenericKernelCallback at 0xff0d2ea7 can be recognized by their strange owner 00004A2A.

```
Type                               Callback   Module              Details
---------------------------------- ---------- ------------------- -------
IoRegisterFsRegistrationChange     0xfc2c0876 sr.sys              -
KeBugCheckCallbackListHead         0xfc1e85ed NDIS.sys            Ndis miniport
KeBugCheckCallbackListHead         0x806d57ca hal.dll             ACPI 1.0 - APIC platform UP
IoRegisterShutdownNotification     0xfc9af5be Fs_Rec.SYS          \FileSystem\Fs_Rec
IoRegisterShutdownNotification     0xfc9af5be Fs_Rec.SYS          \FileSystem\Fs_Rec
IoRegisterShutdownNotification     0xf3b457fa vmhgfs.sys          \FileSystem\vmhgfs
IoRegisterShutdownNotification     0xfc0f765c VIDEOPRT.SYS        \Driver\mnmdd
IoRegisterShutdownNotification     0xfc0f765c VIDEOPRT.SYS        \Driver\VgaSave
IoRegisterShutdownNotification     0xfc6bec74 Cdfs.SYS            \FileSystem\Cdfs
IoRegisterShutdownNotification     0xfc9af5be Fs_Rec.SYS          \FileSystem\Fs_Rec
IoRegisterShutdownNotification     0xfc9af5be Fs_Rec.SYS          \FileSystem\Fs_Rec
IoRegisterShutdownNotification     0xfc9af5be Fs_Rec.SYS          \FileSystem\Fs_Rec
IoRegisterShutdownNotification     0xfc0f765c VIDEOPRT.SYS        \Driver\vmx_svga
IoRegisterShutdownNotification     0xfc0f765c VIDEOPRT.SYS        \Driver\RDPCDD
IoRegisterShutdownNotification     0xfc33d2be ftdisk.sys          \Driver\Ftdisk
IoRegisterShutdownNotification     0xfc1db33d Mup.sys             \FileSystem\Mup
IoRegisterShutdownNotification     0x805f4630 ntoskrnl.exe        \Driver\WMIxWDM
IoRegisterShutdownNotification     0x805cc77c ntoskrnl.exe        \FileSystem\RAW
IoRegisterShutdownNotification     0xfc4ab73a MountMgr.sys        \Driver\MountMgr
GenericKernelCallback              0xfc58e194 vmci.sys            -
GenericKernelCallback              0xff0d2ea7 00004A2A            -
KeRegisterBugCheckReasonCallback   0xfc967ac0 mssmbios.sys        SMBiosDa
KeRegisterBugCheckReasonCallback   0xfc967a78 mssmbios.sys        SMBiosRe
KeRegisterBugCheckReasonCallback   0xfc967a30 mssmbios.sys        SMBiosDa
KeRegisterBugCheckReasonCallback   0xfc0d5006 USBPORT.SYS         USBPORT
KeRegisterBugCheckReasonCallback   0xfc0d4f66 USBPORT.SYS         USBPORT
KeRegisterBugCheckReasonCallback   0xfc0eb3e2 VIDEOPRT.SYS        -
PsSetCreateThreadNotifyRoutine     0xff0d2ea7 00004A2A            -
PsSetCreateProcessNotifyRoutine    0xfc58e194 vmci.sys            -
```

Figure 4.8: Examples of malicious callbacks in a system running BlackEnergy2 malware

4.12 Conclusion

User/kernel memory can provide valuable information about the system's runtime state, particularly when used in conjunction with traditional static and dynamic analysis. Details such as running processes, loaded libraries, logged-in users, listening network sockets, and open files are all available in system memory. We, therefore, analyze the low-level forensic data collected from volatile memory of the sandbox from various aspects and abstract the results in a manner far more abbreviated than the output produced by the existing memory analysis tools. This chapter details most prevalent types of anomalies usually investigated in the manual memory forensics process.

Chapter 5

Implementation

This chapter explains the current implementation of the proposed malware classification framework. The framework has been built upon *Cuckoo* (version 1.1), open source malware analyzer that provides most of the static and dynamic analyses required in the analysis phase. Cuckoo also generates a memory dump of its virtual box on which 20 Volatility commands can be applied. Volatility is an advanced memory forensics tool that analyzes the state of a system using the data found in volatile storage (RAM). We extended Cuckoo's memory analyzer to generate 24 more Volatility commands required in the correlation phase.
To be compatible with Cuckoo, all other modules have been developed in Python. Current implementation of the framework is illustrated in the UML diagram shown in Figure 5.1 followed by the description of the important functions of each class.

5.0.1 Configuration

In the *Config* class, all initial settings required for the proper operation of the framework are defined. Settings can be grouped into: general configuration of the system, knowledge correlator settings, and machine learning parameters. Following is the description of the important functions:

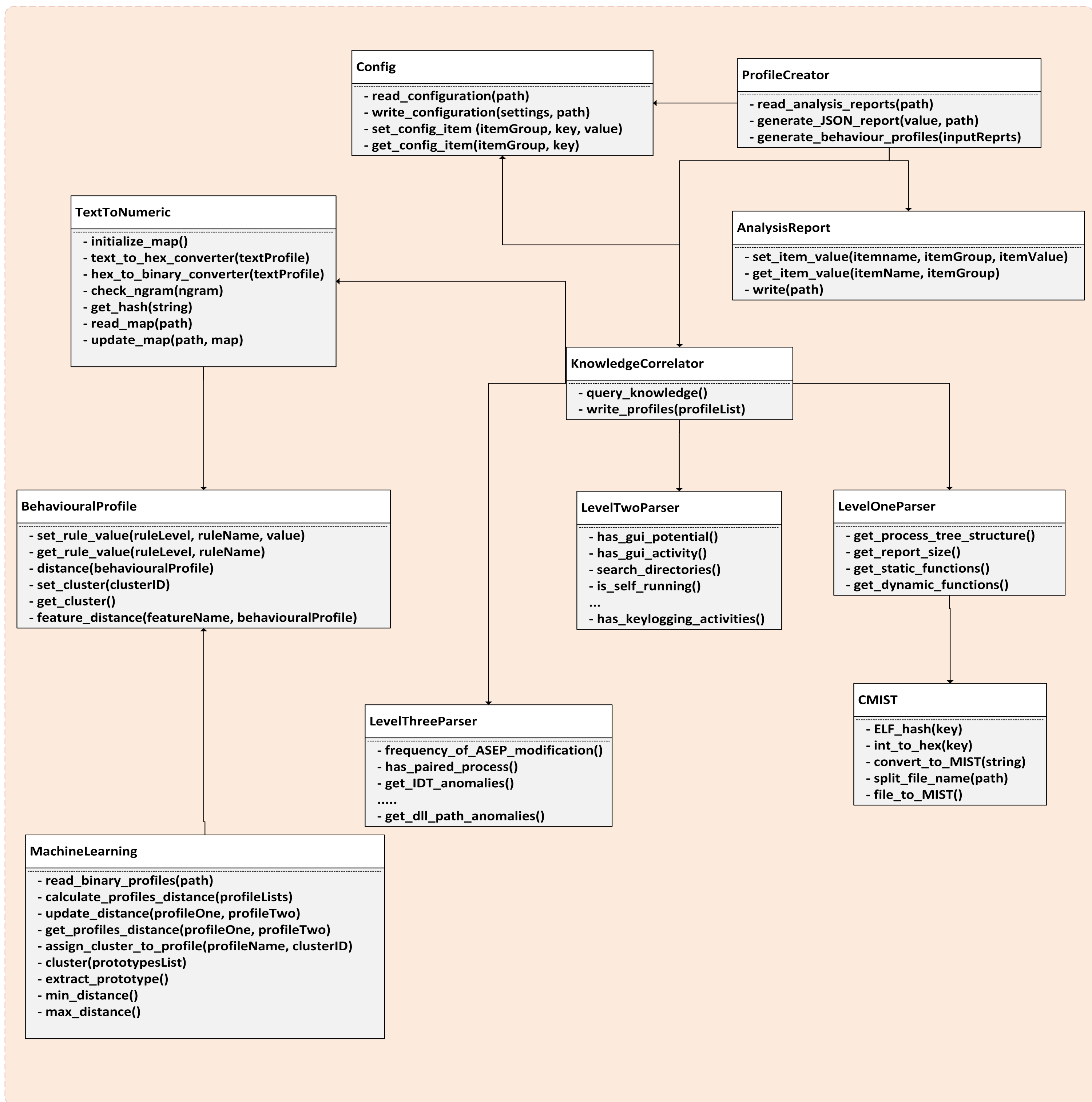

Figure 5.1: UML diagram of the proposed malware classification framework.

- *set_config_item*: sets the value corresponding to the given setting item.
- *get_config_item*: returns the value of the given setting item.
- *read_configuration*: reads the settings into the configuration file.
- *write_configuration*: writes the current settings from the configuration file.

5.0.2 ReportCreator

Acting as the main entry point, this class receives analysis reports from different analyzers and combines them into a single JSON report called analysis report. Main functions of this class include:

- *read_analysis*: reads analysis output generated by different analysis tools.
- *generate_Json_report*: aggregates all analysis information into a single JSON structure (analysis report).
- *generate_behaviour_profiles*: creates a thread for processing each JSON report.

5.0.3 AnalysisReport

This class stores/returns an analysis report. Following is the description of the important functions:

- *set_item_value*: sets the value corresponding to the given item in the JSON report.
- *get_item_value*: returns the value of the given item in the JSON report.
- *write*: writes the analysis into the output directory.

5.0.4 KnowledgeCorrelator

Knowledge correlator connects to the knowledge-base to retrieve information such as known facts about Windows core processes, clean image of IDT/GDT, list of the analyzer's drivers, etc. This information is then distributed to the three *levelOne*, *levelTwo*, and *levelThree* parsers, where a set of rules is checked against the header, API, and memory section of the analysis report. This class includes two main functions:

- *query_knowledge*: reads information from the knowledge-base.
- *write_profiles*: writes all behavioural profiles into the output directory.

5.0.5 LevelOneParser

This class implements basic or level-one rules using information from the header and dynamic analysis result. This class implements the following functions:

- *get_process_tree_structure*: return a tree structure showing the pairwise relationship between child processes.
- *get_report_size*: returns the size of analysis report.
- *get_static_functions*: returns MIST representation of important static functions.
- *get_dynamic_functions*: returns MIST representation of important dynamic functions.

5.0.6 LevelTwoParser

This class implements level-two rules by using sequence of dynamic functions and header information. This class implements the following functions:

- *has_gui_potential*: checks if the program most likely has a GUI.

- *has_gui_activity*: checks if the program displays or manipulate the GUI.
- *search_directories*: checks if the program searches through directories.
- *is_self_running*: checks if the program is able to run itself.
- *low_level_callbacks*: checks if the program uses low-level events as the callbacks.
- *gets_permissions*: checks if the program gets permissions.
- *network_functionality_potential*: checks if the program most likely has network activities.
- *network_activities*: checks if the program has network activities.
- *uses_registry_potential*: checks if the program most likely accesses the registry keys.
- *registers_hotkeys*: checks if the program registers a hot-key combination.
- *executes_programs*: checks if the program executes another program.
- *executes_programs_potential*: checks if the program most likely executes another program.
- *get_packer*: returns the packer found by the signature-based detector.
- *is_packed*: checks different indicators of the packed samples.
- *creates_backdoor*: checks if the program creates a back-door.
- *loads_from_resource_section*: checks if the program loads an embedded program or driver from the resource section of the PE header.
- *modifies_process*: checks if the program modifies another process.
- *has_dynamic_linkage*: checks if the program dynamically imports DLLs or functions.

- *is_persistent*: checks if the program tries to make itself persistent.
- *creates_process*: checks if the program creates another process.
- *loads_dll*: checks if the program loads a DLL.
- *manipulates_files*: checks if the program opens or manipulates files.
- *has_keylogging_activities*: checks if the program has key-logging activity.

5.0.7 LevelThreeParser

This class implements level-three rules using memory dump and dynamic analysis information. Following is the list of main functions:

- *frequency_of_ASEP_modification*: returns a list of excessively accessed/modified registry keys, along with the frequency of access.
- *has_paired_process*: checks if the program has self-healing behaviour.
- *get_IDT_anomalies*: returns IDT anomalies.
- *get_GDT_anomalies*: returns GDT anomalies.
- *is_self_replicator*: checks the type of self-replication behaviour the program may have.
- *get_process_number_anomalies*: returns core processes having abnormal number of instances.
- *get_process_path_anomalies*: returns core processes having abnormal path.
- *get_privilege_enabled_process*: returns a list of process that raised their default privilege.
- *get_orphan_threads*: returns a list of orphan threads.

- *get_threads_hooked_SSDT*: returns a list of threads with hooked SSDT.
- *get_hidden_threads_DKOM*: returns a list of hidden threads using DKOM technique.
- *get_hidden_threads_from_debugger*: returns a list of threads hidden from the debugger.
- *get_threads_with_hardwareBP*: returns a list of threads having hardware breakpoints.
- *get_hidden_unlinked_drivers*: returns a list of unloaded, hidden, and unlinked drivers.
- *get_kernel_hooks*: returns a list of all kernel-level hooks.
- *get_user_hooks*: returns a list of all user-level hooks.
- *has_sensitive_access*: checks if the program accesses sensitive information.
- *has_sensitive_modification*: checks if the program modifies sensitive information.
- *get_drivers_devices_anomalies*: return new drivers, devices, or attached-devices.
- *has_suspicious_timers*: checks if the executable implements a malicious timer.
- *has_suspicious_kernel_callbacks*: checks if executable defines a kernel level callback.
- *has_environment_fingerprinting*: checks existence of different types of fingerprinting methods.
- *get_hidden_dll*: returns hidden DLLs.
- *get_unusual_dll*: returns a list of unusual DLLs imported by the program.
- *get_dll_path_anomalies*: returns DLLs with abnormal path.

5.0.8 CMIST

Current implementation of the MIST presentation has been written for the CWSandbox reports for which source code is not available. Thus, we developed a similar representation compatible with Cuckoo, called CMIST to transform the static and dynamic traces into MIST. This class implements the following functions:

- *ELF_Hash*: calculates the hash value of the input string.
- *int_to_hex*: converts *int* value to the *hex*.
- *convert_to_MIST*: finds MIST value of the input string.
- *split_file_name*: extracts the file name, extension, and the path of the given input.
- *file_to_MIST*: converts the input sequence of static/dynamic functions to MIST.

5.0.9 TesxToNumeric

This class transforms the text behaviour profiles into the binary profiles. The definition of main functions are as follows:

- *initialize_map*: map includes the keys and hash values of each behaviour. This function rests the map.
- *text_to_hex_converter*: finds the hash value of each behaviour and adds it to the map.
- *hex_to_binary_converter*: replaces the hex representation of the behaviour with the sequence of binary digits.
- *check_ngram*: checks if the corresponding behvaiour has given n-gram.
- *get_hash*: finds or calculates the hash value of the input key.

- *read_map*: reads the current map.
- *update_map*: writes new values to the map.

5.0.10 MachineLearning

This class implements *prototype extraction, clustering, and classification* algorithms of the machine learning module. Main functions include:

- *read_binary_profiles*: reads binary behavioural profiles.
- *calculate_profiles_distance*: creates the distance matrix of the profiles.
- *update_distance*: updates the distance between two input profiles.
- *assign_cluster_to_profile*: assigns the profile to its closest cluster.
- *cluster*: clusters the prototypes.
- *extract_prototype*: extracts a set of prototypes from the profiles.
- *min_distance*: returns two profiles with the minimum distance.
- *max_distance*: returns two profiles with the maximum distance.
- *get_profiles_distance*: gets the distance between two input profiles.

5.0.11 BehaviouralProfile

This class implements the attributes and functions required for each behavioural profile. Following is the list of main functions:

- *set_rule_value*: sets the value corresponding to the given rule in the behavioural profile.
- *get_rule_value*: returns the value of the given rule in the behavioural profile.

- *distance*: calculates the euclidean distance from the given profile.

- *set_cluster*: sets the cluster ID of the profile.

- *get_cluster*: returns the current cluster assigned to the profile.

- *feature_distance*: calculates the euclidean distance between two values.

5.1 Conclusion

This chapter presents the design and implementation of the proposed framework. In this architecture, *ProfileCreator* class combines the output of different analyzers into an *AnalysisReport* data structure which is passed to the *knowledgeCorrelator*. The KnowledgeCorrelator sends the report to the three parsers to apply the corresponding rules. Aggregated result is then sent to the *TextToNumeric* class, where a text profile is transformed to a binary profile (in which all values are binary digits). *BehaviouralProfile* object, which represents major behaviours of an executable, is finally classified by the *MachineLearning* module.

Chapter 6

Experiments and Results

To demonstrate the capability of the proposed framework in accurately classifying samples, we compare our framework with two other malware classification approaches. This Chapter starts by describing the reference dataset used for the experiments, and parameters calibrated for the maximum performance of the framework, followed by two evaluation scenarios explained in Section 6.3.

6.1 Datasets and Labeling

Assessing the performance of any detection/classification approach requires test and evaluation with a dataset that is heterogeneous enough to resemble real samples to an acceptable level. We built such dataset by selecting a reasonable mix of different benign and malicious code variants currently popular on the Internet from the following well-known sources:

- *Ether* dataset [20].
- *Malicia* dataset (collected from 500 drive-by download servers over a period of 11 months) [53].
- *VirusTotal* repository [73].

Our dataset contains around 4000 benign executables and 10,000 malware from 264 different families. We added diversity by selecting malware from different categories (viruses, rootkits, etc.), and with different packer. The distribution of the 18 most popular packers observed in our dataset appears in Figure 6.1. Packers with a frequency of less than 10 samples have been grouped under "Other" category.

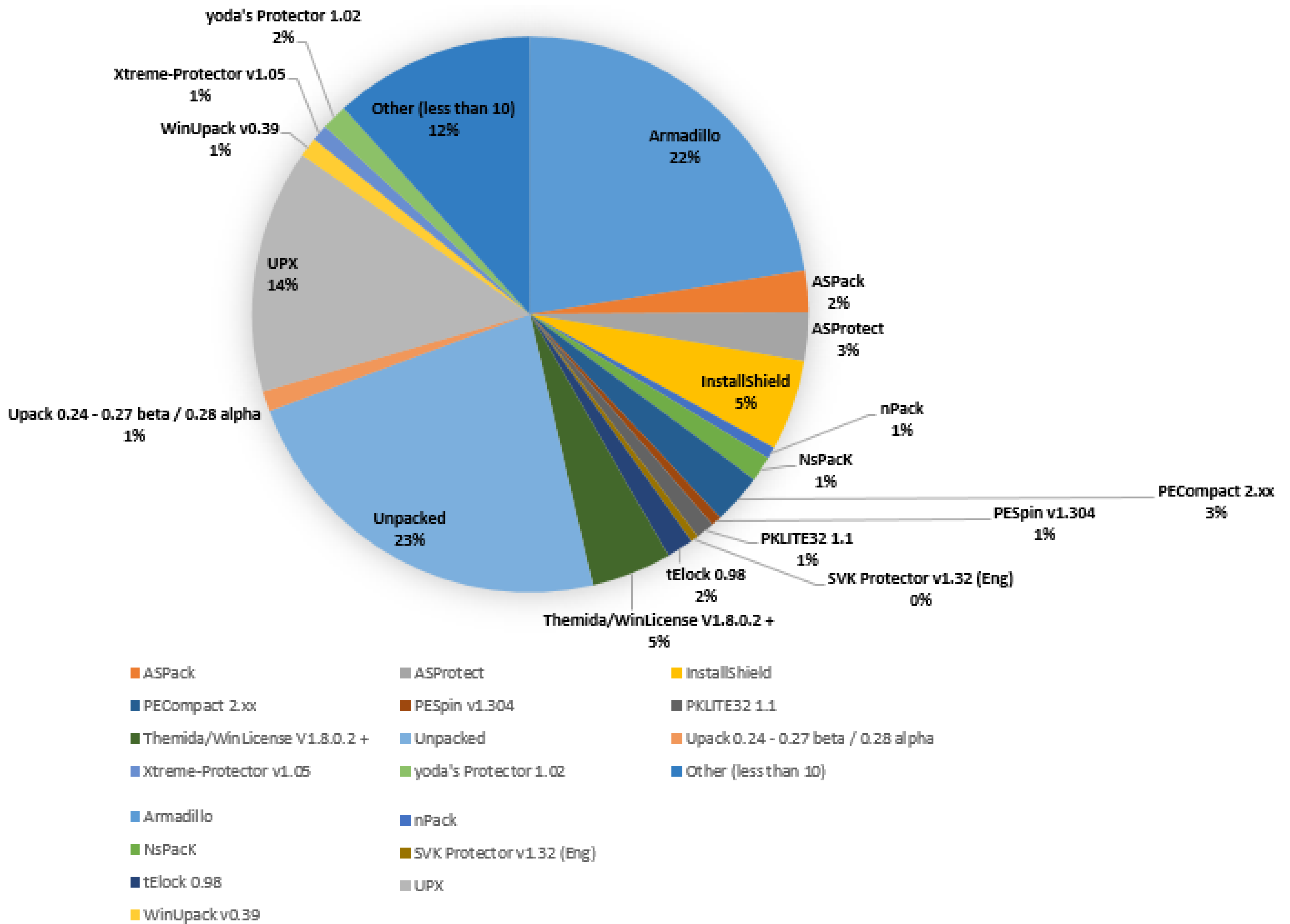

Figure 6.1: Classification performance with regards to different parameters values.

We scanned all samples by 57 online AVs available through Virus Total repository [73] and finally selected Microsoft AV to label the dataset as it was able to successfully label the largest number of samples. Binaries in the dataset were then executed and monitored in

Cuckoo, resulting in a total of 13,000 unique behavioural profiles according to their MD5 value. We identified and discarded 1000 samples (700 malicious and 300 benign) that did not run completely in Cuckoo. Table 6.1 shows the percentage of different malware types in our dataset.

Table 6.1: Statistics on types of malware in the dataset.

Type	**Percentage**	**Type**	**Percentage**
Ransom	1.19%	VirTool	3.31%
Backdoor	9.74%	Tool	4.46%
Rootkits	2.98%	BrowserModifier	1.49%
Virus	1.69%	Rogue	16.99%
Adware	1.53%	Spoofer	1.49%
Trojan (Clicker, Proxy, Spy,...)	19.08%	Worm	1.88%
Dialer	2.98%	MonitoringTool	1.32%
PWS	17.50%	Spyware	1.49%
Exploit	1.99%	Program	4.27%
RemoteAccess	1.49%	Other (less than 1%)	2.48%

Table 6.2 gives a summary of the samples size at each phase. As can be seen, the size of reports generated in the *analysis* phase are extremely large and in some cases exceed 1.5 GB. However, text and binary profiles generated in the *correlation* phase reduce the size of analysis reports 48 and 15 times respectively.

Table 6.2: Size of samples/profiles in MB.

	Min	**Max**	**Average**
Original Samples	0.003	1.59	0.296
Analysis Reports (JSON)	0.013	1584.79	22.25
Text Profiles	0.002	30.98	0.46
Binary Profiles	0.002	2.52	1.46

6.2 System Calibration

The use of machine learning techniques frequently involves careful tuning of learning parameters. Similarly, the discriminating power of the classification phase is significantly influenced by the prototype extraction, clustering and classification algorithms explained in Chapter 5.

These algorithms should be provided with parameters that ensure maximum performance, which in our case means the maximum classification accuracy and the minimum number of rejected or unlabeled reports.

To avoid overtuning, i.e. using all samples to calibrate the machine learning algorithms that fit the training data “too well”, we only use 20% of the dataset for parameter setting and the remainder of the dataset for the final system testing. Following is the list of parameters that need to be optimized:

- max_dist_prototype: Maximum distance between behavioral profiles and existing prototypes.
- min_dist_cluster: Minimum distance between a pair of clusters, i.e. if distance between two clusters is more than min_dist_cluster they are merged.
- max_dist_classify: Maximum distance between each profile and its prototype, i.e. if distance between a profile and its nearest prototype is more that max_dist_classify, that profile is rejected and kept for later incremental analysis.
- reject_num: Clusters having members fewer than reject_num are rejected and kept for later incremental analysis.

Following a brute force search we perform different rounds of experiments to decide on the optimal value for each parameter. Figure 6.2 shows the relationship between the machine learning parameters and classification accuracy. The accuracy is averaged over 10 runs with random initialization of prototype extraction.

As the first component of our ML module, we evaluate the impact of the prototype extraction on accuracy. This process starts by setting an initial value to each parameter, where an acceptable performance is achieved, and continues by only varying the value of max_dist_prototype. By decreasing this parameter from 0.09 to 0.001, maximum accuracy

and minimum number of rejected reports can be achieved (see Figure 6.2), i.e., the smaller we choose the region around the prototypes, the better we can differentiate the known classes of malware. The reason is that a large value for max_dist_prototype causes unknown malware to be incorrectly assigned to prototypes of other malware classes. In these experiments, min_dist_cluster and max_dist_classification are fixed to 0.15 and 0.09 respectively.

We proceed to calibrate the value of min_dist_cluster by using the optimal value of max_dist _prototype (0.001). As can be seen in Figure 6.2, setting too small values (less than 0.1) for min_dist_cluster results in a slight drop in accuracy by creation of more clusters that do not have enough members and are rejected. Figure 6.3 shows the number of rejected reports for different values of min_dist_cluster, where by increasing min_dist_cluster from 0.01 to 0.1, number of rejected reports drops from 215 to 0.

Max_dist_classification does not have any impact on the accuracy at this stage since enough number of representing prototypes is extracted and the distance of each report from its corresponding prototype is less than max_dist_classification. This parameter would affect the number of rejected reports in the stream mode classification.

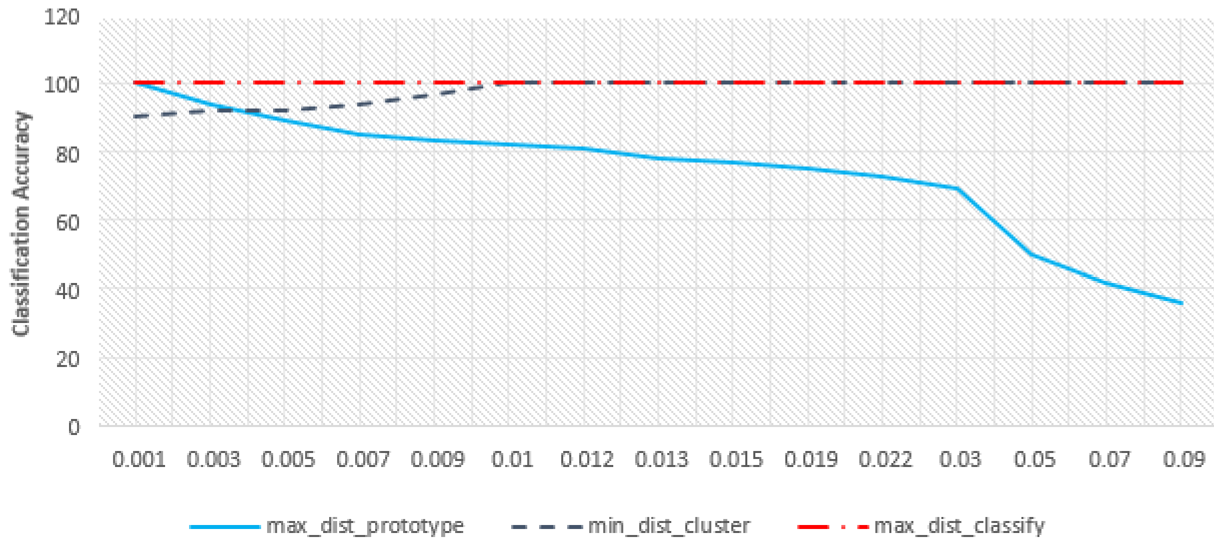

Figure 6.2: Classification performance with regards to different parameters values

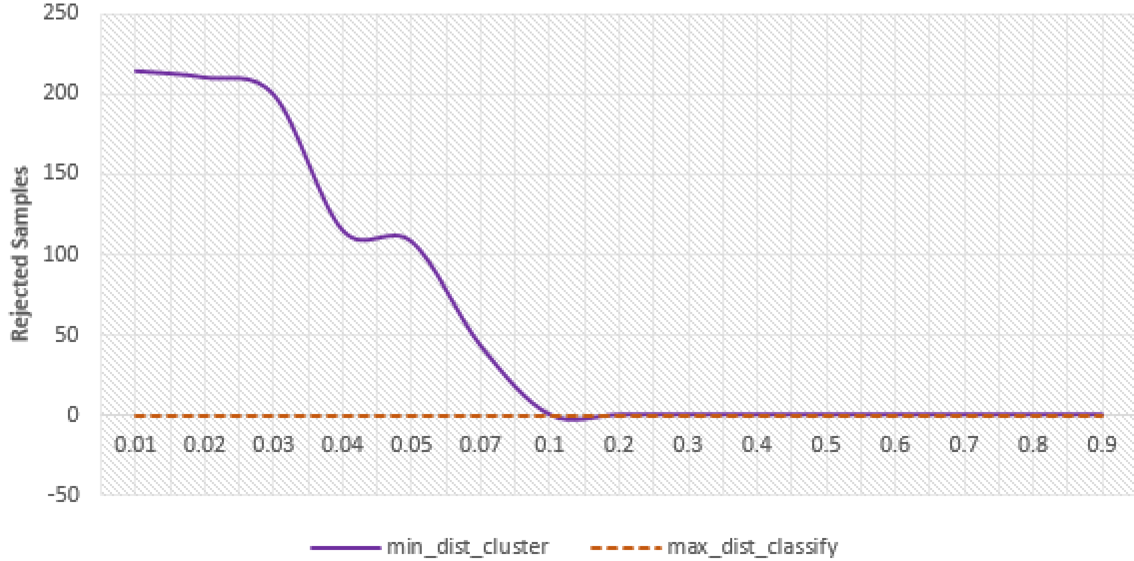

Figure 6.3: Number of rejected reports vs. different values of min_dist_cluster min_dist_cluster.

6.3 Comparative Analysis

To evaluate the effectiveness of the proposed framework, we perform comparison experiments with two other classification approaches: *Malheur* [56] and *Sally* [57].

The purpose of first experiment is to compare the discrimination power of our behvaioural profiles with the features derived by dynamic system call interception. In the second experiment we use the same behavioral profiles and measure the performance of our weighted classification algorithm with Sally, which treats profiles as text documents and performs text classification.

6.3.1 Comparative Evaluation with Malheur

Malheur applies 2-grams analysis on API calls generated by CWSandbox and similar to the proposed framework uses the approximate KNN with Euclidean distance measure to classify analysis reports incrementally. To simulate real-world stream of samples, we randomly split the whole dataset into 6 parts, resembling batches of samples received in 6 consecutive

days. At the end of each day the classification accuracy representing the number of samples correctly labeled is calculated and rejected reports are included in the next day's input samples. Rejected reports includes clusters not having enough members and need more members to be considered as a representative family or samples that their distance from assigned cluster is more than parameter max_dist_classification. Accuracy is calculated in two different ways, excluding rejected samples (ACER) and including the number of rejected reports (ACIR). Figures 6.4 and 6.5 show the classification results achieved on each day.

As can be seen from the Figure 6.4, there are different trends for Malheur and our framework. ACER in the proposed framework increases steadily between day 1 and day 6, from 80% to 89%, while ACER in Malheur follows a descending pattern falling from 90% to less than 82% at day 6. By including rejected samples in the calculation of accuracy, ACIR reflects the performance of each approach more realistically (see Figure 6.5), where our framework keeps its increasing trend while Malheur performance significantly drops.

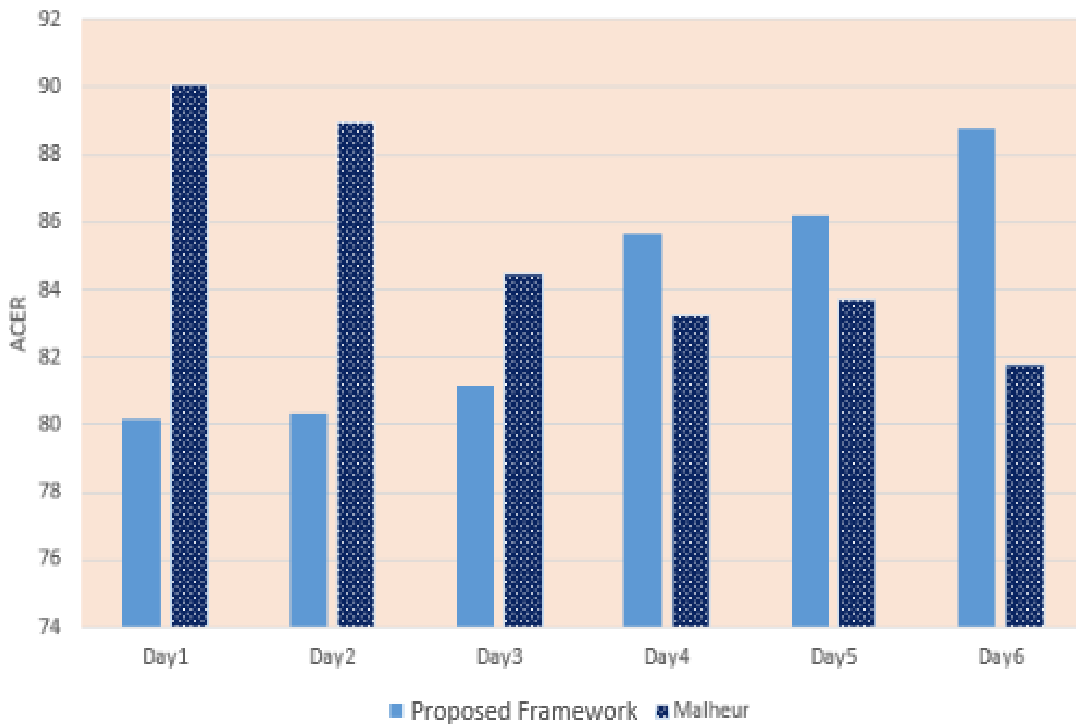

Figure 6.4: Classification performance excluding rejected samples.

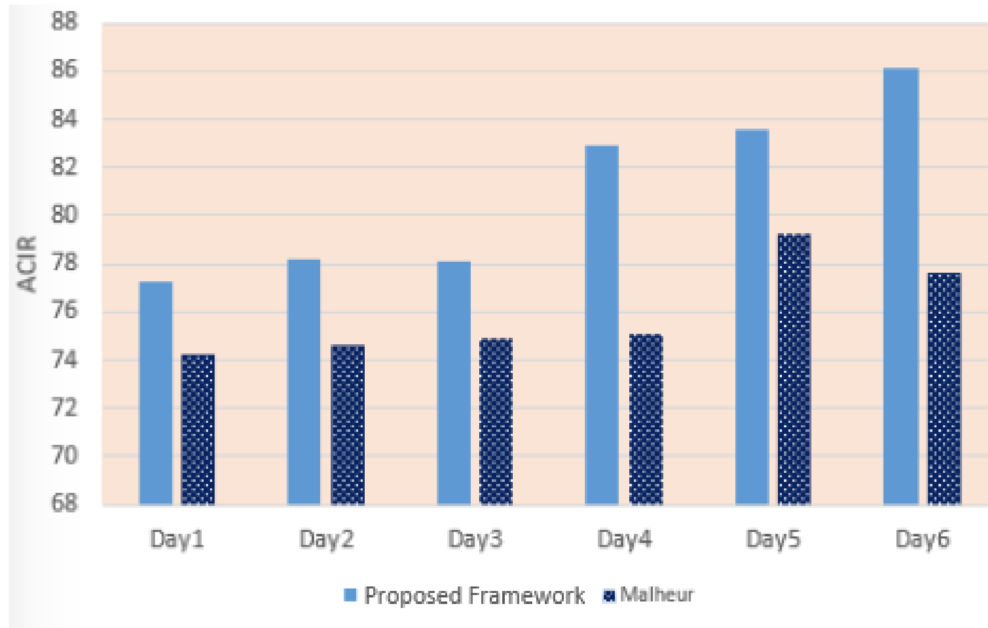

Figure 6.5: Classification performance including rejected samples.

Figure 6.6 shows the number of rejected samples in each approach. For the first two days, the maximum number of profiles are rejected by both approaches, which can be expected since not enough samples are available for most malware families. The big difference in ACER reported by Malheur and our system in these two days can be explained by the fact that Malheur rejects significantly more profiles .

The number of rejected profiles decreases for both approaches as more data is received in the following days. The only exemption happens on day 3, when there is an slight increase in the the number of rejected profiles in our framework (from 57 to 82). One possible explanation can be the increase in the number of new families (more than 20) arrive at this day, for which enough number of instances has not been seen. As can be seen in Figures 6.4 and 6.5, the framework performance starts growing after day 3 as enough number of samples is received for each family and previously rejected profiles are reclustered.

Malheur, on the other hand decreases its rejected profiles from 350 to 244 at day three, however, its ACIR gradually decreases in the following days because of too many falsely

classified samples.

Looking at the most frequent mislabeled samples by Malheur, we observed that benign executables have frequently been classified as *Delf*. Delf is a large family of Trojan horse programs, many of which are associated with data theft. Individual variants of this family have varying characteristics including terminating processes, stealing data, downloading other malware, and performing additional actions without the user's knowledge or permission. We take one of the benign executables as an example and compare its behaviour with a Delf variant to which it was assigned (TrojanProxy:Win32/Delf.W). These two samples are close in terms of invoked API calls. Their distance is less than 0.3 in Malheur scale, where distance is a value between 0 to $\sqrt{2}$. However, they do not show much similarity when comparing their behavioural profiles, especially their level three characteristics. Delf, for instance, has far more user mode hooks, adds new devices to existing drivers, and hooks IDT differently.

Some variants of Delf differ so significantly that they are classified as worms or viruses, rather than Trojans. Comparing behavioural profiles created from two variants of this malware, TrojanDownloader:Win32/Delf.AX and Trojan:Win32/Delflob.I, our high-level rules also show that they are able to differentiate the inter family differences despite their similar characteristics. For example, Delf.AX and Delflob.I share several behaviours; they both search through directories, get permissions, execute other programs, hook IDT, etc. But most of these behaviours are considered level two in our behavioural profiles and have a lower impact on the calculation of distance. In contrast, they have enough level-three characteristics that help in differentiating these two variants. For example, Delflob has self-replication behaviour and has completely different user level hooks.

In general, the large number of misclassifications reported by Malheur results in its descending performance in the following days as depicted in Figures 6.4 and 6.5. Another explanation for the lower accuracy comes from the fact that Cuckoo is not able to capture

system calls, which means many kernel level activities in the samples are discarded in the similarity measurement.

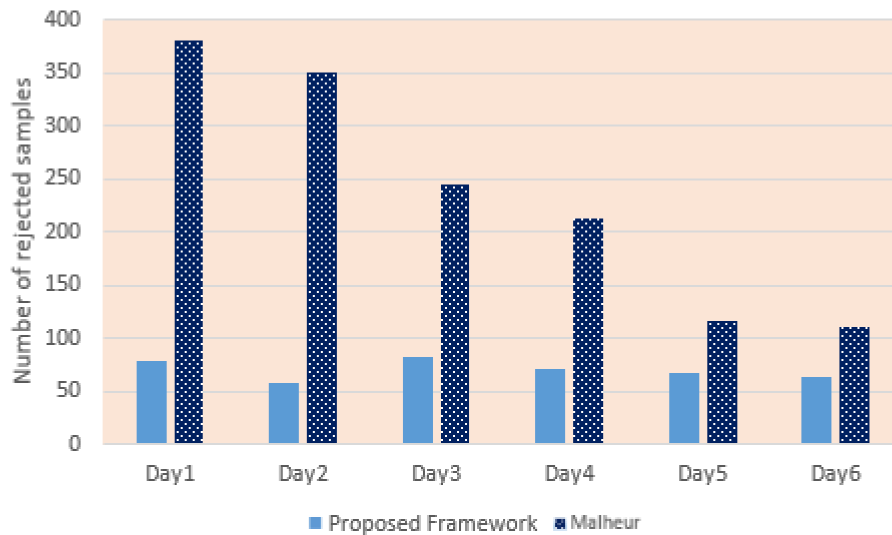

Figure 6.6: Number of rejected samples at the end of each day.

At the end of day 6, the proposed framework classifies samples with the average ACIR of 81% leaving an average of 70 samples unlabeled in each day. In contrast, Malheur, reaches an average accuracy of 76% with an average of 236 unlabeled samples per day.

Table 6.3 gives percentage of samples rejected in each type of malware.

Table 6.3: Details on samples rejected in both approaches.

Type	**Malheur**	**Proposed Framework**	**Type**	**Malheur**	**Proposed Framework**
Ransom	-	4.28%	VirTool	2.11%	7.14%
Backdoor	15.25%	5.71%	Tool	3.38%	4.28%
Rootkits	17.79%	-	BrowserModifier	4.23%	-
Virus	1.69%	5.71%	Rogue	7.2%	4.28%
Adware	-	4.28%	Spoofer	11.44%	5.71%
Trojan (Clicker, Proxy, Spy,...)	5.5%	10%	Worm	-	-
Dialer	2.11%	4.28%	MonitoringTool	4.66%	5.71%
PWS	16.1%	10%	Spyware	-	4.28%
Exploit	2.11%	5.71%	Program	0.84%	8.57%
RemoteAccess	-	4.28%	Other (less than 1%)	5.5%	5.71%

As opposed to our framework, in which rejected samples are uniformly distributed over different types, Malheur shows to be ineffective in classification of some specific types, especially rootkits that mostly have stealthy activities not reflected in user level APIs. Examples of these rootkits are Trojan:Win32/Necurs, Win32/Rustock, and Win32/Cutwail.

We identified a large number of VM-aware malware in samples rejected in our framework, among which are LOIC, BOMBA, and UpClicker trojan, which employ advanced fingerprinting techniques. LOIC and BOMBA display message boxes and wait for the user to click them. UpCliker hooks the mouse by calling the *SetWindowsHookExA* function. These methods of fingerprinting have been defined, and our framework was able to identify some of them while these samples are mostly mislabeled as "benign" in Malheur system.

Total number of each malware type in the dataset also affects the number of rejection. For example, RemoteAccess, Spoofer, or Adware type do not contribute to a big portion of the dataset and some of their instances have been misclassified in the previous days, i.e. not enough representative instances remains at day 6.

6.3.2 Comparative Evaluation with Sally

The first experimental results show the capability of our new behavioural profile in providing comparable classification accuracy compared to that of contemporary dynamic techniques. In the second experiment, we examine our classification approach and method of extracting weighted feature vectors with a standard text classification approach provided by Sally [57]. Sally is a text processing tool for extracting features from strings that are applicable to machine learning algorithms, e.g. n-grams, bytes, tokens. Words (tokens) contained in each document, are considered as a "bag-of-words", where each document is represented by one such bag. Occurrence of tokens in each string are then computed and stored in feature vectors. We then use LibLinear [25], which applies Support Vector Machines (SVM) on a large text dataset.

Table 6.4: Evaluation results

Approach	**Type of learning**	**Profile format**	**Analysis source(s)**	**ACIR**
Proposed framework	Stream	Binary sequence	Static/Dynamic/Memory forensics	81%
Malheur	Stream	MIST	Dynamic	76%
LibSVM	Batch	Text	Static/Dynamic/Memory forensics	67%

Table 6.4 summarizes the average performance of the three approaches. Using the entire dataset with a 5-fold cross validation, LibSVM is able to classify the profiles with an average accuracy of 67%. The low accuracy result reported by the text classification can be explained by considering features as a bag of words and not based on their importance. This results in features with more words (e.g. dynamic traces) having more impact on the classification than some higher level features.

6.3.3 Run-time and Memory Requirements

To have a concise understanding of the time and memory requirements of each approach, we conducted all experiments on an Intel Core i7 processor 3.40 GHz. Although the implementation of our framework provides support for parallelization, we restricted all computations to one core of the processor.

In general, performing memory investigation on the virtualized environment and correlating different analysis results in our framework increase the run-time complexity (approximately 40 seconds per sample) comparing to the two other systems. However, our system requires by far less manual inspection time considering the higher accuracy it provides.

Excluding the intermediate reports, our framework and Malheur have almost the same memory requirement to store reports, as the MIST and binary profiles occupy relatively the same size (between 0.002 to 1.5 MB) on the disk.

Processing time of reports in all three approaches depends on the size of reports. Similar

to Malheur, our framework is able to process an average of 200 reports per second which is sufficient for an large-scale analysis of malware in practice. Processing time of reports in Sally is slightly longer as text reports size vary widely from 0.002 to 31 MB.

Besides the size of reports, the total run-time and memory requirements of each system are affected by the employed classification algorithm. By processing the behavior reports in chunks, the run-time as well as memory requirements of our framework and Malheur are significantly comparable with the batch learning approach implemented by Sally.

6.4 Concluding Remarks

In this chapter we presented the comparative results of applying our framework to a large and diverse dataset of executables with two state-of-the-art classification methods, namely Malheur and Sally. Our first experiment showed that aggregation of different analysis resources (static, dynamic, and memory forensics) brings more insight into malware behaviours comparing to only API calls generated during dynamic analysis. A second experiment was performed with the aim of compared the employed classification algorithm with another recognized machine learning algorithm, SVM. It was found, however, that treating behavioural profiles as text documents results in a significant drop in the classification performance.

The superior classification performance of our framework indicates its effectiveness in stream malware classification.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The inability of signature-based approaches in identifying new malicious symptoms has motivated many researchers to build automated classification approaches using statistical features derived from the static or dynamic analyses. We found that statistical-based approaches are only partially able to distinguish between different malware families. In this research, we have improved an existing machine learning-based classifier by extracting extensive domain knowledge in the form of general and high-level rules. By cross-correlating the results of different analysis tools, we were able to create rules for summarizing an executable's important behaviour in a profile. This profile is more readable for analysts and has higher discriminative power when used in a machine learning algorithm.

In our framework we have focused on the PE header, function calls, and the memory dump of a virtual environment as three sources of data to be matched against our rules. The current implementation of the framework covers eight groups of prominent malicious activities that executables may exhibit. Based on the degree of maliciousness, these behaviours appear in different levels of the behavioual profile and affect the ML algorithm differently. These rules

can be updated over time by finding new information about malware behaviour, but never includes specific characteristics of a malware family.

Our comparative experiment with Malheur [56] shows that the aggregation of different analysis resources, compared to only using API calls, helps differentiate malware families more effectively. Our second experiment also indicates that preserving the importance degree of malware behaviors substantially improves the classification performance. The proposed framework outperforms these two approaches with an average accuracy of 81% and a minimum number of unlabeled samples (0.5% of total samples).

7.2 Future Work

- Experiment with real streams of malware: A future extension would be to collect and analyze more recent and sophisticated samples. We can test our approach and verify the behaviours and characteristics of samples drawn by manual inspection with the output generated by our framework.

- Experiment on a larger dataset: Costly components of the framework should be modified to make it scalable. Parallelization can be performed to provide better run-time performance on common multi-core systems.

- Analysis of other executable types: So far rules written in the knowledge correlator are customized for Windows OS and PE executables. Despite their basic similarities, some characteristics may vary in different OS platforms or malware types. As a future work, analysis and knowledge correlator modules should be extended to support all types of OSs.

- Visualization of behavioural profiles: Current way of abstracting behaviours in the text profiles provides a more convenient way for an analyst to understand the impor-

tant aspects of an executable than the output generated by traditional analyzers, but still can be improved through visualization. Visualization of the behavioural profile can provide easy presentation of malware behaviour and also an easy mechanism for comparing a pair of samples' behaviours.

Bibliography

[1] Dll injection. http://en.wikipedia.org/wiki/DLL_injection. Retrieved August 6, 2014.

[2] Fireeye. http://www.fireeye.com/blog/technical/malware-research/2014/06/turing-test-in-reverse-new-sandbox-evasion-techniques-seek-human-interaction.html. Retrieved July 2, 2014.

[3] Cem Gurkok. Volatility plugin. https://code.google.com/p/volatility/source/browse/trunk/volatility/plugins/linux/check_idt.py?r=3115. Retrieved September 9, 2014.

[4] INFOSEC Institute. Using CreateRemoteThread for DLL Injection on Windows. http://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows/. Retrieved August 3, 2014.

[5] Norman safeground. http://www.norman.com/. Retrieved March 7, 2014.

[6] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Conference on Computer Software and Applications*, volume 2, pages 41–42. IEEE, 2004.

[7] Irfan Ahmed, Aleksandar Zoranic, Salman Javaid, Golden G Richard III, and Vassil Roussev. Idtchecker: Rule-based integrity checking of interrupt descriptor tables in

cloud environments. In *Proceedings of the 9th IFIP WG International Conference on Digital Forensics*, volume 410, pages 305–328. Springer, 2013.

[8] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal of Computer Virology*, 7(4):247–258, 2011.

[9] Michael Bailey, Jon Oberheide, Jon Andersen, Z Morley Mao, Farnam Jahanian, and Jose Nazario. Automated classification and analysis of internet malware. In *Recent advances in intrusion detection*, pages 178–197. Springer, 2007.

[10] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[11] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, and Nadia Tawbi. Static detection of malicious code in executable programs. *International Journal of Requirements Engineering*, pages 184–189, 2001.

[12] Ismael Briones and Aitor Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. In *Virus bulletin conference*, pages 1–12, 2008.

[13] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 122–132. ACM, 2012.

[14] Ero Carrera and Gergely Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Proceedings of 15th virus bulletin international conference*, volume 11, pages 187–197, 2004.

[15] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46. IEEE, 2005.

[16] MNIN Security Blog Coding. Stuxnet's footprint in memory with volatility 2.0. `http://mnin.blogspot.ca/2011/06/examining-stuxnets-footprint-in-memory.html`. Retrieved September 5, 2014.

[17] Cuckoo. Cuckoo. `http://www.cuckoosandbox.org/`. Retrieved March 7, 2014.

[18] Shih-Yao Dai and Sy-Yen Kuo. Mapmon: A host-based malware detection tool. In *13th Pacific Rim International Symposium on Dependable Computing*, pages 349–356. IEEE, 2007.

[19] Microsoft Developer. Irp. `http://msdn.microsoft.com/en-us/library/windows/hardware/ff550694(v=vs.85).aspx`. Retrieved September 4, 2014.

[20] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[21] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects. *SSTIC*, 5:1–3, 2005.

[22] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Xiaodong Song. Dynamic spyware analysis. In *Proceedings of USENIX annual technical conference*, pages 233–246, 2007.

[23] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(6), 2012.

[24] Mojtaba Eskandari and Sattar Hashemi. Ecfgm: enriched control flow graph miner for unknown vicious infected code detection. *Computer Virology*, 8(3):99–108, 2012.

[25] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

[26] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE, 1996.

[27] Teofilo F Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[28] Olivier Henchiri and Nathalie Japkowicz. A feature selection and evaluation scheme for computer virus detection. In *Proceedings of the Sixth ICDM International Conference on Data Mining*, pages 891–895. IEEE, 2006.

[29] HomeFront. Volatility Memory Forensics, Basic Usage for Malware Analysis. `http://www.evild3ad.com/956/volatility-memory-forensics-basic-usage-for-malware-analysis/`. Retrieved August 15, 2014.

[30] Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. Classification of malware based on integrated static and dynamic features. *Network and Computer Applications*, 36(2):646–656, 2013.

[31] Grégoire Jacob, Eric Filiol, and Hervé Debar. Malware as interaction machines: a new framework for behavior modelling. *Journal in Computer Virology*, 4(3):235–250, 2008.

[32] Stephen T Jones, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Vmm-based hidden process detection and identification using lycosid. In *Proceedings of the fourth*

ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pages 91–100. ACM, 2008.

[33] Kad. Handling interrupt descriptor table for fun and profit. http://phrack.org/issues/59/4.html\#article. Retrieved September 9, 2014.

[34] Dae-Ki Kang, Doug Fuller, and Vasant Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Proceedings of 6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW)*, pages 118–125. IEEE, 2005.

[35] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[36] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Detecting malicious code by model checking. In *Proceedings of 2nd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05)*, pages 174–187. Springer, 2005.

[37] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, pages 19–19, 2006.

[38] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[39] McAfee Labs. Mcafee labs threats report. http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q4-2014.pdf. Retrieved May, 2015.

[40] Andrea Lanzi, Monirul I Sharif, and Wenke Lee. K-tracer: A system for extracting kernel malware behavior. In *Proceedings of the 16th Symposium on Network and Distributed System Security*, 2009.

[41] JF Levine, Julian B Grizzard, and Henry L Owen. Detecting and categorizing kernel-level rootkits to aid future detection. *Security & Privacy, IEEE*, 4(1):24–32, 2006.

[42] Zhenkai Liang, Heng Yin, and Dawn Song. Hookfinder: Identifying and understanding malware hooking behaviors. *Proceedings of the 15th Annual Symposium on Network and Distributed System Security*, (41), 2008.

[43] Shun-Te Liu, Hui-ching Huang, and Yi-Ming Chen. A system call analysis method with mapreduce for malware detection. In *Proceedings of 17th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 631–637. IEEE, 2011.

[44] Matt Scuiche. IDTGuard. `http://www.msuiche.net/2006/12/10/idtguard-v01-december-2005-build/`. Retrieved September 9, 2014.

[45] MNIN Security Blog Coding, Reversing, Exploiting. Investigating Windows Threads with volatility. `http://mnin.blogspot.ca/2011/04/investigating-windows-threads-with.html`. Retrieved September 2, 2014.

[46] Eitan Menahem, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Improving malware detection by applying multi-inducer ensemble. *Computational Statistics & Data Analysis*, 53(4):1483–1494, 2009.

[47] Microsoft. Overview of device setup classes. `http://msdn.microsoft.com/en-us/library/windows/hardware/`. Retrieved October 2, 2014.

[48] Microsoft. Secure loading of libraries to prevent dll preloading attacks. `http://support.microsoft.com/kb/2389418`. Retrieved August 11, 2014.

[49] Jose Andre Morales, Peter J Clarke, and Yi Deng. Identification of file infecting viruses through detection of self-reference replication. *Journal in computer virology*, 6(2):161–180, 2010.

[50] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Proceedings of the 23th Annual Conference on Computer Security Applications*, pages 421–430. IEEE, 2007.

[51] Darren Mutz, Fredrik Valeur, Giovanni Vigna, and Christopher Kruegel. Anomalous system call detection. *ACM Transactions on Information and System Security (TISSEC)*, 9(1):61–93, 2006.

[52] Ron oreilly. Ron's vb forum. `http://www.windowsdevcenter.com`. Retrieved September 30, 2014.

[53] MALICIA PROJECT. Malicia project. `http://malicia-project.com/`. Retrieved March 25, 2014.

[54] Yong Qiao, Yuexiang Yang, Jie He, Chuan Tang, and Zhixue Liu. Cbm: Free, automatic malware analysis framework using api call sequences. In *Proceedings of the 10th International Conference on Intelligent Systems and Knowledge Engineering*, pages 225–236. Springer, 2014.

[55] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 108–125. Springer, 2008.

[56] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[57] Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov. Sally: A tool for embedding strings in vector spaces. *The Journal of Machine Learning Research*, 13(1):3247–3251, 2012.

[58] Vino Rosso. Overview of device setup classes. `http://www.systemlookup.com/learn.php`. Retrieved October 2, 2014.

[59] Nasser R. Rowhani. DLL Injection and function interception tutorial. `http://www.codeproject.com/Articles/5178/DLL-Injection-and-function-interception-tutorial`. Retrieved August 6, 2014.

[60] Gajavelly Saichand, M Tech CSE, T Vinay Kumar, and M Tech. Malwise-an effective and efficient classification system for packed and polymorphic malware. *International Journal of Computer Science information and Engineering Technology*, 3, 2014.

[61] Matthew G Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 38–49. IEEE, 2001.

[62] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[63] V Skormin, Alexander Volynkin, D Summerville, and James Moronski. Prevention of information attacks by run-time detection of self-replication in computer codes. *Journal of Computer Security*, 15(2):273–302, 2007.

[64] Victor A Skormin, Douglas H Summerville, and James S Moronski. Detecting malicious codes by the presence of their gene of self-replication. In *Computer Network Security*, pages 195–205. Springer, 2003.

[65] AdlICE Software. Kernelmode rootkits: Part 2, irp hooks. `http://www.adlice.com/kernelmode-rootkits-part-2-irp-hooks`. Retrieved September 4, 2014.

[66] Neuber Software. Common windows processes. `http://www.neuber.com/taskmanager/process/`. Retrieved September 5, 2014.

[67] Sophos. Viruses and spyware. `http://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware.aspx`. Retrieved September 9, 2014.

[68] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In *Malware Detection*, pages 231–249. Springer, 2007.

[69] TechNet. Troubleshooting the startup process. `http://technet.microsoft.com/en-ca/library/bb457123.aspx`. Retrieved August 19, 2014.

[70] Ronghua Tian, Lynn Batten, MR Islam, and Steven Versteeg. An automated classification system based on the strings of trojan and virus families. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 23–30. IEEE, 2009.

[71] Ronghua Tian, Lynn Margaret Batten, and SC Versteeg. Function length as a tool for malware classification. In *Proceedings of the 3rd International Conference on Malicious and Unwanted Software (MALWARE)*, pages 69–76. IEEE, 2008.

[72] Ronghua Tian, MR Islam, Lynn Batten, and Steven Versteeg. Differentiating malware from cleanware using behavioural analysis. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 23–30. IEEE, 2010.

[73] Virus Total. Virustotal-free online virus, malware and url scanner. www.virustotal.com. Retrieved March 7, 2014.

[74] Philipp Trinius, Carsten Willems, Thorsten Holz, and Konrad Rieck. A malware instruction set for behavior-based analysis. In *Proceedings of 5th GI Conference "Sicherheit, Schutz und Zuverl assigkeit"*. German Informatics Society, 2010.

[75] Volatility. An advanced memory forensics framework. `https://code.google.com/p/volatility/wiki/CommandReferenceMal23`. Retrieved July 27, 2014.

[76] Carsten Willems, Thorsten Holz, and Felix Freiling. Cwsandbox: Towards automated dynamic binary analysis. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[77] Ming-Wei Wu, Yi-Min Wang, Sy-Yen Kuo, and Yennun Huang. Self-healing spyware: Detection, and remediation. *IEEE Transactions on Reliability*, 56(4):588–596, 2007.

[78] Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–104. ACM, 2010.

[79] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Proceedings of the 2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300. IEEE, 2010.

[80] Boyun Zhang, Jianping Yin, Jingbo Hao, Shulin Wang, Dingxing Zhang, and WenSheng Tang. New malicious code detection based on n-gram analysis and rough set theory. In *Proceedings of the 2006 International Conference on Computational Intelligence and Security*, volume 2, pages 1229–1232. IEEE, 2006.

Appendix A

Interesting function calls

This appendix contains a list of “interesting Windows functions” from malware analysts’ point of view. This appendix is obtained from the ”Practical Malware Analysis” book [62]. A short description of functions and how they can be used in a malicious way is provided too.

accept Used to listen for incoming connections. This function indicates that the program will listen for incoming connections on a socket.

AdjustTokenPrivileges Used to enable or disable specific access privileges. Malware that performs process injection often calls this function to gain additional permissions.

AttachThreadInput Attaches the input processing for one thread to another so that the second thread receives input events such as keyboard and mouse events. Keyloggers and other spyware use this function.

bind Used to associate a local address to a socket in order to listen for incoming connections.

BitBlt Used to copy graphic data from one device to another. Spyware sometimes uses this function to capture screen shots. This function is often added by the compiler as part of library code.

CallNextHookEx Used within code that is hooking an event set by SetWindowsHookEx.

CallNextHookEx calls the next hook in the chain. Analyze the function calling CallNextHookEx to determine the purpose of a hook set by SetWindowsHookEx.

CertOpenSystemStore Used to access the certificates stored on the local system. CheckRemoteDebuggerPresent Checks to see if a specific process (including your own) is being debugged. This function is sometimes used as part of an anti-debugging technique.

CoCreateInstance Creates a COM object. COM objects provide a wide variety of functionality. The class identifier (CLSID) will tell you which file contains the code that implements the COM object.

connect Used to connect to a remote socket. Malware often uses low-level functionality to connect to a command-and-control server.

ConnectNamedPipe Used to create a server pipe for interprocess communication that will wait for a client pipe to connect. Backdoors and reverse shells sometimes use ConnectNamedPipe to simplify connectivity to a command-and-control server.

ControlService Used to start, stop, modify, or send a signal to a running service. If malware is using its own malicious service, you'll need to analyze the code that implements the service in order to determine the purpose of the call.

CreateFile Creates a new file or opens an existing file.

CreateFileMapping Creates a handle to a file mapping that loads a file into memory and makes it accessible via memory addresses. Launchers, loaders, and injectors use this function to read and modify PE files.

CreateMutex Creates a mutual exclusion object that can be used by malware to ensure that only a single instance of the malware is running on a system at any given time. Malware often uses fixed names for mutexes, which can be good host-based indicators to detect additional installations of the malware.

CreateProcess Creates and launches a new process. If malware creates a new process, you will need to analyze the new process as well.

CreateRemoteThread Used to start a thread in a remote process (one other than the calling process). Launchers and stealth malware use CreateRemoteThread to inject code into a different process.

CreateService Creates a service that can be started at boot time. Malware uses CreateService for persistence, stealth, or to load kernel drivers.

CreateToolhelp32Snapshot Used to create a snapshot of processes, heaps, threads, and modules. Malware often uses this function as part of code that iterates through processes or threads.

CryptAcquireContext Often the first function used by malware to initialize the use of Windows encryption. There are many other functions associated with encryption, most of which start with Crypt.

DeviceIoControl Sends a control message from user space to a device driver. DeviceIoControl is popular with kernel malware because it is an easy, flexible way to pass information between user space and kernel space.

DllCanUnloadNow An exported function that indicates that the program implements a COM server.

DllGetClassObject An exported function that indicates that the program implements a COM server.

DllInstall An exported function that indicates that the program implements a COM server.

DllRegisterServer An exported function that indicates that the program implements a COM server.

DllUnregisterServer An exported function that indicates that the program implements a COM server.

EnableExecuteProtectionSupport An undocumented API function used to modify the Data Execution Protection (DEP) settings of the host, making it more susceptible to attack.

EnumProcesses Used to enumerate through running processes on the system. Malware

often enumerates through processes to find a process to inject into.

EnumProcessModules Used to enumerate the loaded modules (executables and DLLs) for a given process. Malware enumerates through modules when doing injection.

FindFirstFile/FindNextFile Used to search through a directory and enumerate the filesystem.

FindResource Used to find a resource in an executable or loaded DLL. Malware sometimes uses resources to store strings, configuration information, or other malicious files. If you see this function used, check for a .rsrc section in the malware's PE header.

FindWindow Searches for an open window on the desktop. Sometimes this function is used as an anti-debugging technique to search for OllyDbg windows.

FtpPutFile A high-level function for uploading a file to a remote FTP server.

GetAdaptersInfo Used to obtain information about the network adapters on the system. Backdoors sometimes call GetAdaptersInfo as part of a survey to gather information about infected machines. In some cases, it's used to gather MAC addresses to check for VMware as part of anti-virtual machine techniques.

GetAsyncKeyState Used to determine whether a particular key is being pressed. Malware sometimes uses this function to implement a keylogger.

GetDC Returns a handle to a device context for a window or the whole screen. Spyware that takes screen captures often uses this function.

GetForegroundWindow Returns a handle to the window currently in the foreground of the desktop. Keyloggers commonly use this function to determine in which window the user is entering his keystrokes.

gethostbyname Used to perform a DNS lookup on a particular host name prior to making an IP connection to a remote host. Host names that serve as command- and-control servers often make good network-based signatures.

gethostname Retrieves the hostname of the computer. Backdoors sometimes use gethost-

name as part of a survey of the victim machine.

GetKeyState Used by keyloggers to obtain the status of a particular key on the keyboard.

GetModuleFilename Returns the filename of a module that is loaded in the current process. Malware can use this function to modify or copy files in the currently running process.

GetModuleHandle Used to obtain a handle to an already loaded module. Malware may use GetModuleHandle to locate and modify code in a loaded module or to search for a good location to inject code.

GetProcAddress Retrieves the address of a function in a DLL loaded into memory. Used to import functions from other DLLs in addition to the functions imported in the PE file header.

GetStartupInfo Retrieves a structure containing details about how the current process was configured to run, such as where the standard handles are directed.

GetSystemDefaultLangId Returns the default language settings for the system. This can be used to customize displays and filenames, as part of a survey of an infected victim, or by "patriotic" malware that affects only systems from certain regions.

GetTempPath Returns the temporary file path. If you see malware call this function, check whether it reads or writes any files in the temporary file path.

GetThreadContext Returns the context structure of a given thread. The context for a thread stores all the thread information, such as the register values and current state.

GetTickCount Retrieves the number of milliseconds since bootup. This function is sometimes used to gather timing information as an anti-debugging technique. GetTickCount is often added by the compiler and is included in many executables, so simply seeing it as an imported function provides little information.

GetVersionEx Returns information about which version of Windows is currently running. This can be used as part of a victim survey or to select between different offsets for undocumented structures that have changed between different versions of Windows.

GetWindowsDirectory Returns the file path to the Windows directory (usually C: Windows). Malware sometimes uses this call to determine into which directory to install additional malicious programs.

inet_addr Converts an IP address string like 127.0.0.1 so that it can be used by functions such as connect. The string specified can sometimes be used as a network-based signature.

InternetOpen Initializes the high-level Internet access functions from WinINet, such as InternetOpenUrl and InternetReadFile. Searching for InternetOpen is a good way to find the start of Internet access functionality. One of the parameters to InternetOpen is the User-Agent, which can sometimes make a good network-based signature.

InternetOpenUrl Opens a specific URL for a connection using FTP, HTTP, or HTTPS. URLs, if fixed, can often be good network-based signatures.

InternetReadFile Reads data from a previously opened URL.

InternetWriteFile Writes data to a previously opened URL.

IsDebuggerPresent Checks to see if the current process is being debugged, often as part of an anti-debugging technique. This function is often added by the compiler and is included in many executables, so simply seeing it as an imported function provides little information.

IsNTAdmin Checks if the user has administrator privileges.

IsWoW64Process Used by a 32-bit process to determine if it is running on a 64-bit operating system.

LdrLoadDll Low-level function to load a DLL into a process, just like LoadLibrary. Normal programs use LoadLibrary, and the presence of this import may indicate a program that is attempting to be stealthy.

LoadLibrary Loads a DLL into a process that may not have been loaded when the program started. Imported by nearly every Win32 program.

LoadResource Loads a resource from a PE file into memory. Malware sometimes uses resources to store strings, configuration information, or other malicious files.

LsaEnumerateLogonSessions Enumerates through logon sessions on the current system, which can be used as part of a credential stealer.

MapViewOfFile Maps a file into memory and makes the contents of the file accessible via memory addresses. Launchers, loaders, and injectors use this function to read and modify PE files. By using MapViewOfFile, the malware can avoid using WriteFile to modify the contents of a file.

MapVirtualKey Translates a virtual key code into a character value. It is often used by keylogging malware.

MmGetSystemRoutineAddress Similar to GetProcAddress but used by kernel code. This function retrieves the address of a function from another module, but it can only get addresses from ntoskrnl.exe and hal.dll.

Module32First/Module32Next Used to enumerate through modules loaded into a process. Injectors use this function to determine where to inject code.

NetScheduleJobAdd Submits a request for a program to be run at a specified date and time. Malware can use NetScheduleJobAdd to run a different program. As a malware analyst, you'll need to locate and analyze the program that will be run in the future.

NetShareEnum Used to enumerate network shares.

NtQueryDirectoryFile Returns information about files in a directory. Rootkits commonly hook this function in order to hide files.

NtQueryInformationProcess Returns various information about a specified process. This function is sometimes used as an anti-debugging technique because it can return the same information as CheckRemoteDebuggerPresent.

NtSetInformationProcess Can be used to change the privilege level of a program or to bypass Data Execution Prevention (DEP).

OleInitialize Used to initialize the COM library. Programs that use COM objects must call OleInitialize prior to calling any other COM functions.

OpenMutex Opens a handle to a mutual exclusion object that can be used by malware to ensure that only a single instance of malware is running on a system at any given time. Malware often uses fixed names for mutexes, which can be good host-based indicators.

OpenProcess Opens a handle to another process running on the system. This handle can be used to read and write to the other process memory or to inject code into the other process.

OpenSCManager Opens a handle to the service control manager. Any program that installs, modifies, or controls a service must call this function before any other service-manipulation function.

OutputDebugString Outputs a string to a debugger if one is attached. This can be used as an anti-debugging technique.

PeekNamedPipe Used to copy data from a named pipe without removing data from the pipe. This function is popular with reverse shells.

Process32First/Process32Next Used to begin enumerating processes from a previous call to CreateToolhelp32Snapshot. Malware often enumerates through processes to find a process to inject into.

QueryPerformanceCounter Used to retrieve the value of the hardware-based performance counter. This function is sometimes using to gather timing information as part of an anti-debugging technique. It is often added by the compiler and is included in many executables, so simply seeing it as an imported function provides little information.

QueueUserAPC Used to execute code for a different thread. Malware sometimes uses QueueUserAPC to inject code into another process.

ReadProcessMemory Used to read the memory of a remote process.

recv Receives data from a remote machine. Malware often uses this function to receive data from a remote command-and-control server.

RegisterHotKey Used to register a handler to be notified anytime a user enters a particular

key combination (like CTRL-ALT-J), regardless of which window is active when the user presses the key combination. This function is sometimes used by spyware that remains hidden from the user until the key combination is pressed.

RegOpenKey Opens a handle to a registry key for reading and editing. Registry keys are sometimes written as a way for software to achieve persistence on a host. The registry also contains a whole host of operating system and application setting information.

ResumeThread Resumes a previously suspended thread. ResumeThread is used as part of several injection techniques.

RtlCreateRegistryKey Used to create a registry from kernel-mode code.

RtlWriteRegistryValue Used to write a value to the registry from kernel-mode code.

SamIConnect Connects to the Security Account Manager (SAM) in order to make future calls that access credential information. Hash-dumping programs access the SAM database in order to retrieve the hash of users' login passwords.

SamIGetPrivateData Queries the private information about a specific user from the Security Account Manager (SAM) database. Hash-dumping programs access the SAM database in order to retrieve the hash of users' login passwords.

SamQueryInformationUse Queries information about a specific user in the Security Account Manager (SAM) database. Hash-dumping programs access the SAM database in order to retrieve the hash of users' login passwords.

send Sends data to a remote machine. Malware often uses this function to send data to a remote command-and-control server.

SetFileTime Modifies the creation, access, or last modified time of a file. Malware often uses this function to conceal malicious activity.

SetThreadContext Used to modify the context of a given thread. Some injection techniques use SetThreadContext.

SetWindowsHookEx Sets a hook function to be called whenever a certain event is called.

Commonly used with keyloggers and spyware, this function also provides an easy way to load a DLL into all GUI processes on the system. This function is sometimes added by the compiler.

SfcTerminateWatcherThread Used to disable Windows file protection and modify files that otherwise would be protected. SfcFileException can also be used in this capacity.

ShellExecute Used to execute another program. If malware creates a new process, you will need to analyze the new process as well.

StartServiceCtrlDispatcher Used by a service to connect the main thread of the process to the service control manager. Any process that runs as a service must call this function within 30 seconds of startup. Locating this function in malware tells you that the function should be run as a service.

SuspendThread Suspends a thread so that it stops running. Malware will sometimes suspend a thread in order to modify it by performing code injection.

system Function to run another program provided by some C runtime libraries. On Windows, this function serves as a wrapper function to CreateProcess.

Thread32First/Thread32Next Used to iterate through the threads of a process. Injectors use these functions to find an appropriate thread to inject into.

Toolhelp32ReadProcessMemory Used to read the memory of a remote process.

URLDownloadToFile A high-level call to download a file from a web server and save it to disk. This function is popular with downloaders because it implements all the functionality of a downloader in one function call.

VirtualAllocEx A memory-allocation routine that can allocate memory in a remote process. Malware sometimes uses VirtualAllocEx as part of process injection.

VirtualProtectEx Changes the protection on a region of memory. Malware may use this function to change a read-only section of memory to an executable.

WideCharToMultiByte Used to convert a Unicode string into an ASCII string.

WinExec Used to execute another program. If malware creates a new process, you will need to analyze the new process as well.

WlxLoggedOnSAS (and other Wlx* functions) A function that must be exported by DLLs that will act as authentication modules. Malware that exports many Wlx* functions might be performing Graphical Identification and Authentication (GINA) replacement.

Wow64DisableWow64FsRedirection Disables file redirection that occurs in 32-bit files loaded on a 64-bit system. If a 32-bit application writes to C:\Windows \System32 after calling this function, then it will write to the real C:\Windows\System32 instead of being redirected to C:\Windows\SysWOW64.

WriteProcessMemory Used to write data to a remote process. Malware uses WriteProcessMemory as part of process injection.

WSAStartup Used to initialize low-level network functionality. Finding calls to WSAStartup can often be an easy way to locate the start of network related functionality.

Appendix B

List of common auto-run registry keys

Registry Key	Description
• *HKCU\Software\Microsoft\Windows\CurrentVersion \Explorer\ ShellFolders\Startup*	Contains path to the start up programs.
• *HKLM\Software\Microsoft\Windows\CurrentVersion \Run*	Allows a program to be automatically executed at system startup [52].
• *HKCU\Software\Microsoft\Windows\CurrentVersion \Run*	Allows a program to be automatically executed at system startup, and when a specific user is logged in [52].
• *HKLM\Software\Microsoft\Windows\CurrentVersion \RunOnce*	Allows a program to be automatically executed at next system startup. The key is then being removed [52].
• *HKCU\Software\Microsoft\Windows\CurrentVersion \RunOnce*	Allows a program to be automatically executed next time the user logs in. The key is then being removed [52].

Registry Key	Description
• *HKLM\Software\Microsoft\Windows\CurrentVersion \RunServices*	Allows a service to be automatically started at system startup [52].
• *HKLM\Software\Microsoft\Windows\CurrentVersion \RunServicesOnce*	Allows a service to be automatically started at next system startup. The key is then being removed [52].
• *HKLM\Software\Microsoft\Windows\CurrentVersion \ShellServiceObjectDelayLoad*	Entries specified under this key are loaded at the system startup.
• *HKLM\Software\Microsoft\WindowsNT \CurrentVersion\Winlogon*	
• *HKLM\Software\Microsoft\Windows\CurrentVersion \RunEx* • *HKLM\Software\Microsoft\Windows\CurrentVersion \RunOnceEx*	
• *HKCU\Software\Microsoft\WindowsNT \CurrentVersion\Windows\run*	
• *HKCU\Software\Microsoft\WindowsNT \CurrentVersion\Windows\load*	
• *HKLM\Software\Microsoft\Windows\CurrentVersion\ Policies\Explorer\Run*	Items listed under this key (usually malicious) are added to the SystemLookup Startup list [52]
• *HKCU\Software\Microsoft\Windows\CurrentVersion\ Policies\Explorer\Run*	Items listed under this key (usually malicious) are added to the SystemLookup Startup list [52]

Registry Key	Description
• *HKLM\SYSTEM\CurrentControlSet\Control\Class\{ClassGuid}*	For most devices, there is a setup class (e.g. Image, Battery or Keyboard setup class) defined by Microsoft. If a new device wants to use one of the existing setup classes it can refer to the GUID of the corresponding setup class [47]. For example, *HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E96B − E325 − 11CE − BFC1 − 08002BE10318}* defines setup class for keyboard devices. Using this key a nefarious keylogger driver can be loaded same as a benign keyboard driver [77].
• *HKLMNT_Dlls*	Allows a DLL to be loaded into every process that links with User32.dll.
• *HKLM\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters\Protocol_Catalog9* • *HKLM\SYSTEM\CurrentControlSet\Services\WinSock2\Parameters\NameSpace_Catalog5*	Can be used to load a *Layered Service Provider*[1] Dll into all process that import networking components.
• *HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Windows\AppInit_DLLs*	AppInit Dlls defined under this registry key are loaded with all process that use user32.dll [58].
• *HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\Notify\{dllname}*	Dlls defined under this key are used to handle Winlogon.exe events [58]. Some spyware, e.g. CoolWebSearch, use this feature to replace Windows Services with their own malicious Dlls [77].
• *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\SharedTaskScheduler*	Entries under this key are CLSIDs[2] that are loaded when the computer starts. File information of each key is available under *HKLM\SOFTWARE\Classes\CLSID\{CLSID}* key [58].
• *HKLM\SYSTEM\CurrentControlSet\Services*	Runs a program as a service whenever system starts up. Malware can use this key to either register a new autostart service or modify an existing security related service.

[1] A Layered Service Provider is a DLL that places itself between the Internet and an application to intercept or modify the traffic going inside or outside [58]

[2] A globally unique identifier for a COM class object.

Registry Key	**Description**
• *HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\Explorer\ShellExecuteHooks*	Similar to Shared Task Scheduler, entries under this key point to a CLSID, which is loaded when system starts up [58].
• *HKLM\SOFTWARE\Microsoft\CodeStoreDatabase \DistributionUnits{CLSID}*	ActiveX controls or add-ons[3] defined under this key point to a CLSID that contains file information. Malicious ActiveX add-ons (not necessarily to create an ASEP) may be installed while viewing a website [58].
• *HKLM\Software\Microsoft\InternetExplorer\Toolbar* • *HKCR\Protocols\Name − SpaceHandler* • *HKLM\Software\Microsoft\InternetExplorer\Search \SearchAssistant* • *HKLM\Software\Microsoft\InternetExplorer\Search \CustomizeSearch* • *HKCU\Software\Microsoft\InternetExplorer\Main \SearchBar* • *HKCU\Software\Microsoft\InternetExplorer\ SearchURL* • *HKCU\Software\Microsoft\InternetExplorer\Main \SearchPage* • *HKCU\Software\Microsoft\InternetExplorer\Main \Default_Page_URL* • *HKCU\Software\Microsoft\InternetExplorer\Styles \UseMyStylesheet* • *HKCU\Software\Microsoft\InternetExplorer\Main \StartPage* • *HKLM\Software\Microsoft\InternetExplorer\ URLSearchHooks* • *HKLM\Software\Microsoft\Windows\CurrentVersion \Explorer\BrowserHelperObjects* • *HKLM\Software\Microsoft\InternetExplorer\ ExplorerBars*	These registry keys can be used to attach a Browser Helper Object (BHO) entry, toolbar, Dll, or a URL into Internet Explorer [77, 58].

Table B.1: Auto start registry keys

[3] Small programs that improve IE

Appendix C

Windows core processes

Process	Known Facts	Reference
System.exe	• Runs Windows kernel • Has more than one instances	[69] [66]
Smss.exe	• Starts user session, Winlogon.exe, and Csrss.exe services • Sets system variables	[69] [66]
lsass	• Is the local security authentication subsystem server that authenticates users and generates access token • Is created after startup and by Winlogon.exe (or Wininit) • Has normally one instance • Its default path is $c:\backslash Windows\backslash system32$	[16] [66] [69]

Process	Known Facts	Reference
csrss	• Is the client/server run-time subsystem responsible for handling console windows, creating and deleting threads • Is an always running subsystem process • Its default path is $c:\backslash Windows\backslash system32$	[66] [69]
spoolsv	• Handles spooled print and fax jobs • Its default path is $c:\backslash Windows\backslash system32$	[66] [69]
alg	• Is an application layer gateway service • Manages connections to third-party firewalls or Internet Connection Sharing • Its default path is $c:\backslash Windows\backslash system32$	[66] [69]
svchost	• Is an essential part of the OS on which other processes (those invoked by Dll) are run • May have more than one instances • Is responsible to load services defined by the registry on start up • Its default path is $c:\backslash Windows\backslash system32$	[66] [69] [29]

Process	Known Facts	Reference
services	• Is the service control manager that starts, stops or changes the default setting of services • Its default path is $c:\backslash Windows\backslash system32$	[66] [69]
winlogon	• Handles user logon or logoff requests • Its default path is $c:\backslash Windows\backslash system32$	[66] [69]
wmiprvse	• Multiple instances of it can be run under LocalSystem, NetworkService or LocalService accounts • Its default path is: $C:\backslash WINDOWS\backslash System32\backslash Wbem$	[66] [69]
rundll32.exe	Is used to load a dll but seeing this process in the process list is suspicious	[77] [69]

Table C.1: Known facts about Windows OS core processes

Vita

Candidate's full name: Elaheh Biglar Beigi Samani

University attended:

Master of Computer Science
University of New Brunswick
2013-2015

Bachelor of Information Technology
Isfahan University of Technology
2006-2010

Publications:

Elaheh Biglar Beigi, Hossein Hadian Jazi, Natalia Stakhanova, and Ali A Ghorbani. Towards effective feature selection in machine learning-based botnet detection approaches. In *Proceedings of the 2014 IEEE Conference on Communications and Network Security (CNS)*, pages 247-255. IEEE, 2014.

Fakhroddin Noorbehbahani, Elaheh Biglar Beigi Samani, and Hossein Hadian Jazi. A novel method for learner assessment based on learner annotations. *Journal of Educational Technology & Society*, 16(3):88-101, 2013.

Sayed Hadi Hashemi, Mohammad Babaeizadeh, Mohsen Nowruzi, Hossein Hadian Jazi, Mohammad Shahmoradi, and Elaheh Biglar Beigi Samani. A comprehensive semi-automated incident handling workflow. In *Proceedings of the 6th International Symposium on Telecommunications (IST)*, pages 1065-1070. IEEE, 2012.