ISKRA: BARE-METAL WINDOWS MALWARE DYNAMIC ANALYSIS FRAMEWORK

by
YUSUF ARSLAN POLAT

Submitted to the Graduate School of Cyber Security
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
Sep 2020

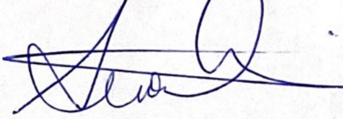
ISKRA: Bare-Metal Windows Malware Dynamic Analysis Framework

APPROVED BY:

Prof. Dr Erkay SAVAŞ
(Thesis Supervisor)

Assoc. Prof. Dr. Cemal YILMAZ

Asst. Prof. Dr. Ahmet Onur DURAHİM



DATE OF APPROVAL: 08/09/2020

YUSUF ARSLAN POLAT 2020 ©

All Rights Reserved

ABSTRACT

ISKRA: BARE-METAL WINDOWS MALWARE DYNAMIC ANALYSIS FRAMEWORK

YUSUF ARSLAN POLAT

CYBER SECURITY MSc THESIS, SEP 2020

Thesis Supervisor: Prof. Dr. ERKAY SAVAS

Keywords: malware, hypervisor, sandbox, dynamic analysis, evasion

With the proliferation of "cyber-crime as a service" economy, besides gaining new victims, providing permanence on them has been one of the key points of profit for attackers. Thus, hiding malicious presence while operating is now more important for malware than being fully undetectable when it is first distributed. Due to the increasing number of malware attacks[1] and prohibitively long hours required for manual inspection, analysts often use dynamic analysis platforms to investigate malware samples. However, these platforms have been repeatedly shown to fail to combat evasion methods that are constantly updated by attackers[2] (Jadhav, Vidyarthi & Hemavathy M., 2016). Even if malware is correctly classified by the existing dynamic analysis platforms, which are widely deployed in the cyber security industry, it has been frequently observed that the malware detects the analysis environment and behaves differently to evade inspection; consequently the malicious code targeted by the attacker does not execute. In this case, the inspection, which will make the malicious code run and be examined, has to be done by the analyst manually. In this study, we present the bare metal hypervisor-based framework for dynamic analysis, ISKRA, which facilitates system calls to be collected and analyzed without being detected by malware. ISKRA is a portable and easily modifiable framework and not only allows any system to be easily transformed into an analysis environment, regardless of the virtual machine or bare metal; but also allows for forensics to be

[1]See https://www.av-test.org/en/statistics/malware/

[2]See https://www.watchguard.com/wgrd-resource-center/security-report-q4-2019

run without being detected in live systems. This way, incident response specialists can quickly transform the system under inspection into an analysis environment and can collect evidence, examine and remedy the system without being detected by the attacker. We designed, implemented and experimented with the framework, which employs machine learning algorithms to learn from new attack campaigns. Our work shows that the framework leads to negligibly low overhead and provides a high detection rate for the most current malware campaigns that evade dynamic inspection by other frameworks.

ÖZET

ISKRA : DİNAMIK ZARARLI YAZILIM ANALİZİ PLATFORMU

YUSUF ARSLAN POLAT

SİBER GÜVENLİK YÜKSEK LİSANS TEZİ, EYLÜL 2020

Tez Danışmanı: Prof. Dr. ERKAY SAVAŞ

Anahtar Kelimeler: zararlı yazılım, dinamik analiz, kum havuzu, hipervizör, antivirüs atlatma

Siber saldırganların yer aldığı yeraltı ekonomisinde siber suç servisleri yaygınlaşmıştır. Bu yeni ekonomik sistemde saldırganların yeni kurbanlar elde etmesinin yanı sıra hali hazırda erişim elde ettikleri kurban sistemler üzerinde erişimlerini korumaları da saldırganlar için son derece önem arz eden bir duruma gelmiştir. Geçmiş dönemde zararlı yazılımların saldırının ilk anında güvenlik ürünleri tarafından tespit edilemez olmaları önemliyken, artık sistemi ele geçirdikten sonra da bu tespit edilemezliklerini korumaları gerekmektedir. Sayıları gitgide artan ve elle (İng. manual) incelenmeleri bir hayli vakit alan bu zararlı yazılımları incelemek için analistler genellikle dinamik analiz platformlarını kullanmaktadırlar. Ancak bu platformlar, saldırganlar tarafından sürekli yenilenen/güncellenen/iyileştirilen *atlatma* yöntemleri nedeniyle yetersiz kalmaktadırlar. Zararlı yazılımlar, bu platformlar içerisinde analiz edildiklerini tespit edebilmekte ve gerçek amaçlarını gizlemek üzere davranışlarını değiştirebilmektedirler. Bu nedenle dinamik analiz platformları zararlı yazılımları başarı ile sınıflandırabilseler dahi zararlı yazılımın gerçek davranışını gözlemleyemedikleri vakalar oluşmaktadır. Platformların bu yetersizlikleri nedeniyle analistler gün sonunda yine elle analize mecbur kalmaktadırlar. Bu çalışmada sunduğumuz ISKRA isimli hipervizör tabanlı dinamik analiz platformu, fiziksel bilgisayar üzerinde zararlı yazılımlar tarafından tespit edilmeden sistem çağrılarının toplanmasına ve zararlı yazılımların analiz edilmesine imkân sağlamaktadır. Kolay kurulabilir ve değiştirilebilir olan bu platform fiziksel ve sanal sistemlerde çalışabilmesinin yanı sıra hali hazırda çalışan bir sistemi analiz ortamına dönüştürebilmektedir. Böylelikle çalışan bir sistem üzerinde canlı olay müdahalesi yapılmasına imkân sağlamaktadır. Dolayısıyla olay müdahale ekipleri vaka yaşanan sistemi analiz or-

tamına dönüştürüp zararlı yazılım tarafından tespit edilmeden delil toplama, inceleme ve tedavi yapabilmektedirler. Çalışmamız kapsamında öne sürdüğümüz platformu geliştirdik ve makina öğrenmesi ile yeni zararlı yazılım saldırılarını tespit etmek üzere deneyler gerçekleştirdik. Gerçekleştirdiğimiz deneyler platformumuzun düşük sistem yükü ve yüksek doğruluk oranıyla diğer dinamik analiz platformlarının tespit edemediği güncel zararlı yazılım saldırılarını tespit edebildiğini göstermiştir.

ACKNOWLEDGEMENTS

This study was made possible by the support of a few people for whom I am grateful. First of all, I would like to thank my supervisor, Professor Erkay Savas.

I thank my family for their support during the thesis process.

to my mother, father and, brother

TABLE OF CONTENTS

LIST OF TABLES ..... **xii**

LIST OF FIGURES ..... **xiii**

1. INTRODUCTION **1**

2. BACKGROUND **5**
- 2.1. Malware on Windows 5
  - 2.1.1. A Brief Study of Malware for Windows Operating System 6
  - 2.1.2. Current Threat Landscape 7
- 2.2. Current Malware Analysis and Detection Methodologies 7
  - 2.2.1. Static Analysis Based Detection 8
  - 2.2.2. Dynamic Analysis Based Detection 8
- 2.3. Machine Learning for Malware Analysis and Detection 9
- 2.4. Evasion Techniques 10
  - 2.4.1. Hardware Based 11
  - 2.4.2. Software Based 12
- 2.5. Technologies Used To Create Analysis Environment 15
  - 2.5.1. Virtualization Technologies 15
    - 2.5.1.1. Intel VT-x 16

3. ISKRA FRAMEWORK **20**
- 3.1. Data Collection 21
- 3.2. Data Preprocessing 22
- 3.3. Threat Report Generation 24

4. BEHAVIOUR DATA MONITORING **25**

5. MODEL CONSTRUCTION **27**

6. EXPERIMENTAL RESULTS **31**
- 6.1. Real World Examples 31

6.1.1. Zloader 32
6.1.2. GuLoader 34
6.1.3. MassLogger 35
6.1.4. Cradle Ransomware 37
6.2. Samples from VirusTotal 39
6.2.1. String Parameter Classification 43
6.3. Benchmark 45

7. RELATED WORK 47

8. DISCUSSION 49

9. CONCLUSION AND FURTHER WORK 51

BIBLIOGRAPHY 52

LIST OF TABLES

Table 5.1. An Example Subset Of The System Calls And Their Unique Integer Values 27
Table 5.2. Hooked system calls and their corresponding categories 28

Table 6.1. Case I: classification results for all system calls using monograms (i.e., $n = 1$) and dictionary size is 22 41
Table 6.2. Case II: classification results for system calls in the sample data set using monograms (i.e., $n = 1$) and dictionary size is 21 41
Table 6.3. Case I: results for n-gram range of [2, 4] and dictionary size is 3725 42
Table 6.4. Case II: results for n-gram range of [2, 4] and dictionary size is 3899 42
Table 6.5. Tested hyperparametes for the LSTM networks 43
Table 6.6. LSTM experiments results for Case II 43
Table 6.7. Classification scores of string system call parameters using various machine learning algorithms (Case II and $n = 3$) 44
Table 6.8. String classification using decision tree algorithm (Case II and $n \in [2, 10]$) 44
Table 6.9. Test Environment System Specification 45
Table 6.10. Benchmark Results 45

Table 7.1. Comparison of ISKRA and known and popular dynamic analysis platforms. 47

LIST OF FIGURES

Figure 2.1. Hardware Information Based Sandbox Detection Method Used By Gozi Malware 11
Figure 2.2. Registry Configuration Based Sandbox Detection Method Used By Kutaki Malware 14
Figure 2.3. Inline hooking overview 18

Figure 3.1. Overview of ISKRA 21
Figure 3.2. Captured System Call Example 23

Figure 4.1. Overview of kernel component 26

Figure 5.1. Example API Sequence for Code Injection 29

Figure 6.1. Zloader Network Traffic Captured by the ISKRA 33
Figure 6.2. GuLoader Network Traffic Captured by the ISKRA 35
Figure 6.3. MassLogger Network Traffic Captured by ISKRA 37
Figure 6.4. MassLogger Configuration Extracted from the ISKRA logs. 38
Figure 6.5. Ransom Note Obtained with the ISKRA. 39
Figure 6.6. System Calls Count Variation 40

1. INTRODUCTION

Malware, a widely used abbreviation for malicious software that has become a part of our lives since the late 1980s, is used to describe any piece of software developed to damage computer systems, usually for some benefit to the attacker (Milošević, 2014). Malware, originally developed to demonstrate talent in the hacker sub-culture when it is first introduced, is one of the most essential tools of cyberattacks. When it was only a part of the sub-culture, it used to suffice to detect malware by using traditional signature-based methods. Today, however, much more sophisticated and evasive malware samples are used as a part of more extensive operations such as APT (Advanced Persistent Threat)[1] attacks. Therefore, much more detailed information is needed about current malware, such as its functions, how it operates, how it communicates with its command and control server, and the traces it leaves in the system such as files and registry keys.

Static and dynamic analysis are the two main approaches used by analysts to investigate malware behaviour and/or structure (Sikorski & Honig, 2012). The static analysis refers to the examination of suspicious software using reverse engineering without running its harmful code. Nowadays, anti-analysis methods such as obfuscation, packing[2], and encryption are widely and skilfully deployed in malware by attackers to increase the time analysts spend on static reverse engineering; already a very time-consuming process (Moser, Kruegel & Kirda, 2007). This situation contradicts with the primary aim of analysts: to detect and prevent an attack campaign in the shortest time possible as it is typically a race against time between analyst and attacker. As a result, it has been observed that static analysis is generally used not to perform the entire analysis, but to examine specific suspicious parts in an executable file.

The dynamic analysis approach, on the other hand, is based on monitoring the behavior of malware in a controlled environment. During the dynamic analysis, the

[1] See https://www.kaspersky.com/resource-center/definitions/advanced-persistent-threats

[2] See https://www.blackhat.com/docs/us-14/materials/us-14-Mesbahi-One-Packer-To-Rule-Them-All-WP.pdf

analyst can interfere with the behavior of the malware with tools such as debuggers or observe the routine behavior in the controlled environment without interfering. Due to the immense workload instigated by increased malware attacks nowadays, analysts generally employ automatic dynamic analysis of malware in a controlled environment and examining its runtime behavior therein. The collection of tools that allow automatic dynamic analysis of harmful software are referred as *sandbox*. These tools collect events such as file and registry events that may be related to malware while it is running and submit them to analyst. Then, analyst decides necessary actions in a short period of time based on these events. Thus, nowadays, analysts heavily depend on sandbox tools for the inspection of suspicious software.

Attackers, on the other hand, have developed various anti-analysis methods against sandbox solutions as well as against static analysis[3,4] (Lundsgård & Nedström, 2016). Often, these methods are based on profiling the system, in which malware executes, and distinguishing the sandbox environment. As analysts first use sandbox to expeditiously examine numerous number of malware attacks, decision for further actions is made based on the findings from the sandbox report. Therefore, malware's ability to hide itself at this stage ensures that the attack can continue for a much longer time. This diminishes the reliability of the sandbox outputs and necessitates manual inspection and verification of sandbox outputs and therefore removes the advantages of sandbox to a large extent due to associated long delays in manual analysis. For this reason, sandboxes should be impervious to evasive malware so that the analysis can be done accurately and in a short time.

In this article, we present the ISKRA dynamic analysis platform that enables automatic analysis in bare-metal systems and evades detection by malware. Unlike the methods used in many traditional sandboxes, the platform is developed employing *inline hooking* supported by Extended Page Tables (EPT) and Intel VT-x virtualization technologies. It collects mainly operating system calls by processes while minimizing traces of its presence and detects malicious software via machine learning algorithms performed on system call sequences and their relevant string parameter values.

Existing sandbox solutions usually depend on specific virtualization technologies. In an incident response case, a specific virtual machine and/or guest operating system should first be installed in the computer system to perform the sandbox analysis. If the system, where the incident occurs, has specific software or hardware requirements, then they have to be reproduced in these virtual environment as well.

---

[3]See https://2018.hack.lu/archive/2014/Bypasss_sandboxes_for_fun.pdf

[4]See https://bit.ly/32gzCZ1

Only after that, it is possible to examine malicious software targeting that system. Our proposed framework ISKRA can be installed on any computer system with VT-x technology using only one installation file and allows the transformation of an already running system to an analysis environment. Thus, it shortens the time required to install a new system. In addition, this framework, which hides itself from all software running on the operating system at the higher level of privilege by using Intel VT-x technology, provides the opportunity to collect evidence without being detected by malicious software.

We used our framework to collect information about process system calls and apply various machine learning algoithms to detect malware samples in our experimental setup. We used the Area Under Curve (AUC) score of Receiver Operating Characteristic (ROC) to evaluate machine learning experiments. Our results show that 82% AUC is obtained for a data set consisting of 40 harmful and 32 benign software samples. In addition, we selected some malware attacks, which are active in the first half of 2020 and can detect and bypass existing dynamic analysis platforms. We showed that malware samples used in those attacks execute their malicious payload in ISKRA and thus give in to detailed investigation afterwards.

The contributions of our work can be summarized as follows:

- We designed and implemented a dynamic analysis framework ISKRA based on the Intel VT-x with EPT technology and showed its advantages for the inspection of evasive malware.
- We compared the efficiency of different machine learning and deep learning algorithms for the use of system call sequences and their parameters in malware analysis.
- We demonstrated the efficiency and effectiveness of our approach with current malware samples.
- We emphasized the risks and complications associated with evasive malware in incident response operations and showed that our approach can be an effective solution.

The remainder of this thesis is structured as follows: Chapter II presents background knowledge relevant to the field of this study. Chapter III describes the ISKRA dynamic analysis framework and outlines our its design principals and details. Chapter IV provides detailed information about our framework's behaviour data monitoring methodology. Chapter V describes the constructed machine learning model with the malware behaviour data, which is collected by the proposed

framework. Chapter VI shares the results of the experiments performed on the most recent malware attacks with the proposed framework. Specifically, it provides and evaluates the results of the various machine learning models and discusses the overhead in system performance due to the framework, which is shown to be negligible. Chapter VII refers to the studies in the literature related to the subject and includes comparative analysis between those works and the proposed framework. Chapter VIII discusses the immunity of the ISKRA to evasion techniques employed by evasive malware. Chapter IX concludes the thesis and proposes ideas for future work.

2. BACKGROUND

2.1 Malware on Windows

Malware, or malicious software, is a term used for referring any software that intentionally harms a computer system. They may have been developed for different purposes such as fame or profit for the attacker behind the malware. They are divided into types such as adware, virus, worm, trojan, stealer, loader, rootkit, and ransomware depending on their purpose and techniques employed. We provide the definitions of various malware in the following.

Adware is used to refer to all unwanted software that hide themselves in the system and show advertisements to users.

Virus is a type of malware that modifies other executable files and replicates itself into them.

Worm is malicious software that spreads itself into other computer systems.

The main difference between worm and virus is that the former interacts with other computer systems and replicates itself into them. Worm samples generally exploit vulnerabilities in network services to copy their infected executable files into other computers. So they need no user interaction for them to be spread and executed. But, virus samples follow a much passive strategy of spreading themselves. They need a third party user or program to copy and run an infected executable in a new victim computer. So, infected executable in the new machine can infect other files in that system. Also, worms tend to spread more quickly than viruses as they have no dormant stage as in the case of viruses.

Trojan is a malware type that enables an attacker to control the victim host.

Stealer is a type of malicious software that is used to steal user data from the victim host. They generally target registered passwords in the system and banking credentials of victim users.

Loader is a type of software used for deploying new malware in the victim machine. Most of the cases, they have no other malicious intentions (such as stealing credentials, self-replicating, etc.) other than downloading and executing new malware that does the harm.

Rootkit is a type of malware that executes in the kernel-mode of an operating system. They are generally used for hiding artifacts, which are created by themselves or by other malware of the same attacker, from users by altering data structures of the operating system. For example, they can alter the linked list of the running processes in the kernel to hide from any program that lists processes such as task manager.

Ransomware is a type of malware that encrypts user files in victim machines using public key cryptography. The decryption key is only known by the attacker. Later, attacker requests money from the victim in exchange of the decryption key.

2.1.1 A Brief Study of Malware for Windows Operating System

Microsoft Windows, also referred to as Windows OS or shortly Windows, is a personal computer (PC) operating system (OS) developed by Microsoft Corporation. Its first version was released in the mid-1980s and its many different versions have been released since then. The current version is Windows 10 at the time of writing. Windows is the most popular operating system for desktop PCs as it approximately accounts for the share of 77.74% of all desktop operating systems[1]. Due to its popularity, it is widely targeted by attackers that aim to obtain a great number of victims. An independent cybersecurity organization states that 78% of new malware samples

[1] https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

target Windows systems in 2019[2].

2.1.2 Current Threat Landscape

Modern malware campaigns are overly complicated to be developed and managed by ordinary attacker groups. Therefore, similar to Software as a Service (SaaS) in the legitimate business community, attackers started to use Cybercrime as a Service (CaaS) solutions (Manky, 2013). Threat actors buy or rent malware executables, command and control servers (C&C), and any other necessities for cyber attacks from the services in the underground attacker community. Consequently, the same malware from the same CaaS is used by many different threat actors. These trends seem to make it easier to create signatures for malware campaigns from analysts' points of view. Nevertheless, CaaS providers implement evasion methods in their malware to hide new attack campaigns and preserve persistence on victims. Otherwise, their malware would be unusable in a short amount of time. We observe a rapid growth in evasive malware campaigns[3] and therefore discern that one of the most pressing concerns in cybersecurity is fast and robust detection and analysis of evasive malware.

2.2 Current Malware Analysis and Detection Methodologies

In this section, we discuss the current methods, techniques and approaches commonly used for malware detection and/or analysis. We will inspect mainly static, dynamic and machine learning-based detection and/or analysis approaches.

---

[2] https://www.av-test.org/en/news/facts-analyses-on-the-threat-scenario-the-av-test-security-report-2019-2020/

[3] https://embed.widencdn.net/download/watchguard/gyjc30tyxq/Threat_Report_Q4_2019_Overview.pdf

2.2.1 Static Analysis Based Detection

The static analysis pertains to techniques for analyzing malware samples without executing them (Uppal, Mehra & Verma, 2014). The samples are analyzed by using the static reverse-engineering methodology. Opcodes (operation code, also known as instruction code) of the executables are extracted and examined. They are usually converted to human-readable formats by using disassembling (translates opcodes into assembly language) and decompilation (translates opcodes into high-level languages such as C) methods. Most of the malware analysis is usually performed manually, which is a highly time-consuming process as it requires a detailed code review carried out on low-level code samples. Also, anti-analysis methods such as obfuscation, packing, encryption of executable resources are used by attackers, which make static analysis even more time-consuming. Consequently, rather than being used as a sole analysis method, static analysis is mostly used to gain more insight about an observation captured during dynamic analysis. For instance, if analysts observe an encrypted file in the dynamic analysis, they perform static analysis to reveal the encryption algorithm, its keys, and plain data.

2.2.2 Dynamic Analysis Based Detection

The dynamic analysis refers to techniques for examining malware samples by executing them in a controlled environment and monitoring their actions (Gadhiya & Bhavsar, 2020). There are two different approaches in dynamic analysis: advanced manual and automated. In the advanced manual analysis, analysts use assembly level debuggers (e.g., OllyDbg, Windbg, x64Dbg, etc.) to perform dynamic code analysis as the source code of executable is not available. This approach is useful to examine only a small segment of code. However, it has similar limitations as in the case static code analysis since it also requires code analysis on low-level codes. In the second approach, analysts automatically execute and monitor activities in a controlled environment. This approach requires no manual code analysis. Analysts easily observe behaviors of executable to contemplate and take necessary actions against malware attacks. The approach is easy to deploy as an analyst needs no high-level reverse engineering skills. Also, because of its automated nature, it allows analyzing more samples in short amount of time when compared to other analysis approaches.

As discussed in the previous section, the number of malware attacks is growing rapidly. Also, attack motivation and techniques are changing frequently. Therefore, solutions that can quickly learn from attacks and thus adapt to rapidly changing malware attack techniques and strategies are needed. Recently, employing machine learning algorithms is a frequently used approach to meet these needs in the field of cybersecurity due to their ability to automatically learn and improve through experience. In this section, we will discuss machine learning algorithms that are commonly used in the field of cybersecurity.

Logistic Regression is a predictive analysis method to model a dichotomous (binary) dependent variable. It is used for classification problems with two possible values[4]. It is different from the linear regression in the sense that it is used for classification whereas the latter solves regression problems.

Support Vector Machines(SVMs) are a set of supervised learning models used for solving classification, regression analysis, and outlier detection problems. They are used to classify binary and multi-class datasets[5]. They represent data points in a hyperspace and classify them by finding an optimal hyperplane[6] in the hyperspace that separates classes from each other.

k-nearest neighbors is a supervised learning algorithm that is used to solve classification and regression problems[7]. It predicts the class of a new data point by the selecting the most common class label among its k nearest neighbors in the training data set[8].

Decision Tree is a supervised learning method that uses a model resembling a tree, in which each internal node represents a test on the value of an attribute and the branch does the outcome of the test. Each leave in the tree is in fact a class label and a path from the root of the tree to a particular leaf is a

[4]See https://christophm.github.io/interpretable-ml-book/logistic.html

[5]See https://scikit-learn.org/stable/modules/svm.html

[6]See https://mathworld.wolfram.com/Hyperplane.html

[7]See https://scikit-learn.org/stable/modules/neighbors.html

[8]See http://www.robots.ox.ac.uk/ dclaus/digits/neighbour.htm

decision rule. The tree is first formed from the labeled data points in the training data set. Then in the test phase, decision rules are applied on attributes of new data points and their class labels are assigned accordingly. Decision trees are used frequently for solving classification and regression analysis[9].

Random Forest is a learning method used for regression analysis and classification employing more than one decision tree. The prediction for class label is then obtained by combining results from a number of individual decision trees[10]. The model mitigates overfitting problem that is common in decision trees and thus improves the model accuracy by using the mean of the predictions from the individual trees or their most common prediction.

Long Short-Term Memory(LSTM) is a deep learning algorithm and type of artificial recurrent neural network (RNN) architecture[11]. The network is able to learn long-term dependencies in time series data. The system calls by malware can be considered as time-series data as they are invoked in a sequential manner. LSTM networks are well-suited to keep track of long-term dependencies of system calls that identifies malicious system calls, between which there are time lags of unknown duration.

2.4 Evasion Techniques

Evasion techniques describe methods used by malicious software to hide themselves and their activities from cybersecurity products such as dynamic analysis platforms. In this section, we discuss common evasion methods. These techniques are essentially based on identifying the characteristics of the current working environment and comparing them with characteristics of common end-user systems. The basic assumption is that dynamic analysis platforms deviate from common end-user platforms in many aspects. The system characteristics that are examined by malware can be classified as software and hardware characteristics.

---

[9] See https://scikit-learn.org/stable/modules/tree.html

[10] See https://www.shirin-glander.de/2018/10/ml_basics_rf/

[11] See https://missinglink.ai/guides/neural-network-concepts/deep-learning-long-short-term-memory-lstm-networks-remember/

2.4.1 Hardware Based

In this section, we will discuss the approaches used by malware to evade security products by using the hardware features of the execution environment.

Hardware Information Common virtualization solutions such as Xen, QEMU, and KVM that are used by existing dynamic analysis solutions have no intention to hide their presence. They use their product signatures in simulated hardware. For example, the VMware virtualization software use "00:50:56", "00:0C:29", and "00:05:69" in the first three bytes of its MAC address[12]. These three bytes are OUI (organizationally unique identifier), which uniquely distinguishes the vendor of the network card. Consequently, malware can use that signature to identify a VMware based dynamic analysis platform. Figure 2.1 shows a code snippet from the leaked Gozi [13] malware source project that is used for detecting sandboxes by using device names. It compares device names in the execution environment with the name of common virtualization products.

Figure 2.1 Hardware Information Based Sandbox Detection Method Used By Gozi Malware

Hardware Implementation Differences Virtualization software basically simulates physical systems. In order to imitate new technologies developed in hardware products, the virtualization software code needs to be updated. However, there can be implementation differences between physical and virtual systems. For

---

[12] See https://nakedsecurity.sophos.com/2016/12/13/nymaim-using-mac-addresses-to-uncover-virtual-environments-and-bypass-antivirus/

[13] See https://malpedia.caad.fkie.fraunhofer.de/details/win.isfb

example, virtual systems use different locations for Interrupt Descriptor Table[14]. In some cases, certain features are never added to virtualization software due to their development costs. For example, the MMX instruction set, which is used for faster graphical processing, is not implemented in VMware[15]. Consequently, malware can use all these implementation differences for detecting a virtualized execution environment.

2.4.2 Software Based

In this section, we will present the approaches used by malware to detect and evade security solutions by using the software characteristics of the execution environment.

Analysis Tools Windows is an object-based operating system. The essential features of the operating system are accomplished through these objects. For example, a file object (defined as FILE_OBJECT structure) represents an instance of a file, device, directory, or volume and it is essential to interact with a file in the operating system[16]. There are more than 25 types of objects in the Windows operating system such as file, process, registry, and devices[17].

Evasive malware enumerates objects in the current running environment to find any artifact related to malware analysis tools.

File System The installed programs, running applications, and processed data in the operating system leave traces (such as installation archives, executable files, page cache, etc.), which can be volatile or permanent, in the file system. Malicious software checks the presence of the traces to detect analysis tools. For instance, RTM banking malware check the following files and directories in the victim system; and if any of those exists, malware accepts the system as a sandbox and stops execution[18]:

- C:\cuckoo

[14]See https://www.lions.odu.edu/ c1wang/course/cs495/lecture/10_2_Anti-VM_Techniques.pdf

[15]See https://www.cyberbit.com/anti-vm-and-anti-sandbox-explained/

[16]See https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/ns-wdm-_file_object

[17]See https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-object-manager

[18]See https://unit42.paloaltonetworks.com/russian-language-malspam-pushing-redaman-banking-malware/

- C:\fake_drive
- C:\perl
- C:\strawberry
- C:\targets.xls
- C:\tsl
- C:\wget.exe
- C:*python*

Process In short, a process is an executing program. The operating system loads an executable file and its dependencies in a process memory space and executes its instructions. Malware analysts use various types of programs to interact with infected systems and examine running malware; those include debuggers, system monitors, network packet analyzers, etc. On the other hand, malware utilizes process properties of widely used analyst tools and sandboxes to detect analysis environments. For instance, a ransomware campaign gets the list of running processes in the target machine and checks the existence of the following process names in the list to detect Joe Sandbox, Sandboxie, and other analyst tools[19]:

- sandboxierpcss.exe
- sandboxiedcomlaunch.exe
- procmon.exe
- joeboxserver.exe
- apimonitor.exe
- behaviordumper.exe

The full list can ben found at the analysis report.[20]

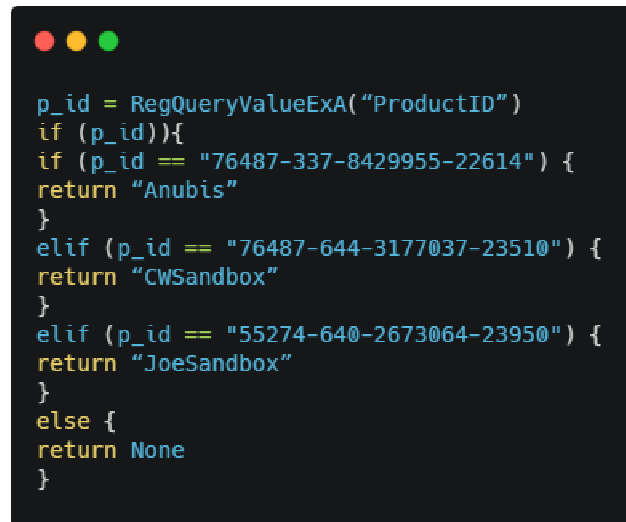
Registry The Windows Registry is a system-defined database that is used by the Windows operating system and applications to manage their configurations. Both user mode and kernel mode system components can use the registry to save and retrieve settings. Malware analyst tools also use the Registry to transform the operating system into an analyst environment and save their settings. Malware

[19]See https://securelist.com/to-crypt-or-to-mine-that-is-the-question/86307/

[20]See footnote 19

seeking for commonly used analysis environment and tool settings to distinguish sandboxes. Figure 2.2 shows a code snippet from the Kutaki malware that detects Joe Sandbox, CWSandbox, and Anubis by checking the "ProductID" configuration in the "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion" registry key[21]

Figure 2.2 Registry Configuration Based Sandbox Detection Method Used By Kutaki Malware



Environment Differences Besides hardware specifications (as discussed in the previous section), dynamic analysis solutions also need to simulate common usage patterns of operating systems and applications. Otherwise, they can be easily distinguished by malware. For instance, malware obtains the list of MRU (Most Recently Used) files in the current environment from Windows registry. Based on the file count in the MRU list, malware validates that the current environment is not a sandbox platform as there is real user interaction with the system that leaves artifacts (such as registry values and files[22]) on the operating system similar to an end-user system[23].

[21] See https://cofense.com/kutaki-malware-bypasses-gateways-steal-users-credentials/

[22] See https://www.andreafortuna.org/2017/10/18/windows-registry-in-forensic-analysis/

[23] See https://www.mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/

2.5 Technologies Used To Create Analysis Environment

Malware analysts need environments to execute malware samples and monitor their behaviors. These environments should satisfy some conditions for safe and effective analysis. First of all, they should be similar to the target system of malware. Thus, malware samples should run without any software incompatibility issue (one cannot run a x64 compiled executable on x86 analysis environment) and therefore evasion attack surface will be reduced (See Section 2.4). Secondly, various types of malware can have different execution requirements such as specific hardware as observed in Stuxnet attacks[24]. Therefore, it should be easy to adapt analysis environments to meet the requirements of a malware sample. Finally, due to the potential risks of the malware attacks, every second counts in the malware analysis and incident response. Therefore, analysis environments should be easy to setup and fast to run. Security researchers use virtualization technologies to create analysis environments that answer all these requirements. In the rest of this section, we will discuss common virtualization technologies and Intel VT-x used in this study.

2.5.1 Virtualization Technologies

Virtualization refers to creating a virtual representation of any system, such as computer hardware, architecture and computer networks[25].

Software-Based Virtualization is a method that executes the guest system by emulating it in software[26]. Virtualization software examines instructions that originate from the guest system before executing them and detect privileged instructions. They are executed after the detected instructions are replaced with safe equivalents. This procedure is called binary translation[27]. VMware workstation (with 32-Bit guest OS) and VirtualBox (again with 32-bit guest OS) are examples

[24]See https://www.wired.com/images_blogs/threatlevel/2010/11/w32_stuxnet_dossier.pdf

[25]See https://www.vmware.com/tr/solutions/virtualization.html

[26]See https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-B14C8267-C2A4-4BF8-B680-70C2B350B325.html

[27]See https://blogs.oracle.com/ravello/nested-virtualization-with-binary-translation

of the virtualization solutions that use software-based approach[28].

Hardware-Assisted Virtualization method uses the support of the technologies that are already built-in the processors. Thus, it eliminates the necessity of binary translation and improves virtualization performance[29]. Its implementation by Intel is known as Intel VT-x.

Software or hardware that run virtual machines is also called a *hypervisor*. There are two types of hypervisors based on their interaction with computer hardware: type 1 and type 2. Type 1 hypervisors are directly executed on the hardware and they are also called bare-metal hypervisors. Type 2 hypervisors are executed on a host operating system instead of computer hardware[30]. Because of their direct communication with computer hardware, bare-metal hypervisors have performance advantage over type 2 hypervisors (Ganesan, Murarka, Sarkar & Frey, 2013).

2.5.1.1 Intel VT-x

In 2005 and 2006, two major semiconductor chip manufacturers (Intel and AMD) independently proposed new instruction set extensions to their x86 architectures. Intel named it as Intel VT-x (Virtualization Technology) and AMD used the term AMD Virtualization (AMD-V) for its similar extensions. Both of the solutions aim to create a hypervisor that runs operating systems without any modification (e.g., binary translation) and performance losses[31]. In our study, we opt for Intel processors due to their larger market share[32]. However, a similar solution can also be developed for the systems using AMD processors. This calls for a different study, which is outside the scope of this work.

The Virtual Machine Extensions (VMX) mode concept is one of the core components of these improvements. The VMX mode allows the desired code to run in a virtual environment while preserving the integrity of the host system. Also, it does not

[28] https://www.unixarena.com/2017/12/para-virtualization-full-virtualization-hardware-assisted-virtualization.html/,

[29] See footnote 28

[30] See https://phoenixnap.com/kb/what-is-hypervisor-type-1-2

[31] See https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf

[32] See https://www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/

require heavy software development efforts such as binary translation because it is already implemented on the processor by default. Thus, virtualization is now a more easily accessible technology. It is possible to take advantage of these features of the VT-x technology by executing the necessary assembly instructions in a running bare-metal system. For example, a researcher created a hypervisor with 10 lines of assembly code[33].

After the initial release, Intel proposed additional improvements to its virtualization technology such as the Extended Page Table (EPT) feature. EPT aims to avoid the overhead of page table operations incurred due to the translation between the virtual machine and hypervisor. To this end, it allows a guest operating system to modify directly its own page tables. Consequently, the virtual machine can control memory operations (read, write, and execute) in the guest machine thanks to its control over the page-table[34].

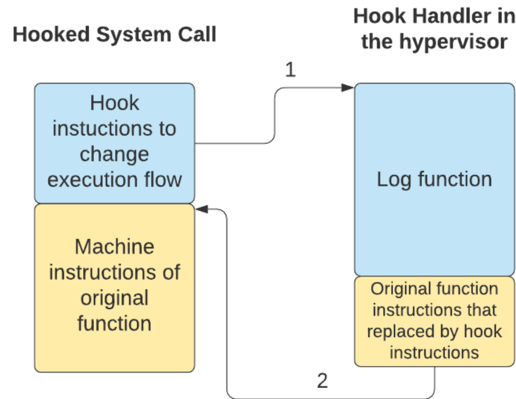
As a result of all these improvements, Intel provides a complete virtualization solution through VT-x for all users. Indeed, existing tools for dynamic malware analysis take advantage of these benefits using VT-x technology through large-scale projects such as Xen. They use these advantages to emulate the entire operating system in a virtualized environment. This makes the tools of the current solutions both challenging to deploy and enables them to be detected by evasive malware due to their differences from bare-metal systems. In our study, we use bare-metal (physical system) directly as the analysis environment. Also, we use VT-x technology not to emulate the operating system, but to execute our analysis tools in a more privileged level than the operating system. Thus, we aim to have more control over the operating system kernel and running applications. However, as mainstream dynamic analysis tools run on top of a virtualization layer the latter needs it to be installed first, which is impossible for a running system. Also, this additional layer may incur time and resource overhead, which can create anomaly, which helps malware detect the analysis tool. ISKRA suffers none of those disadvantages.

When ISKRA starts working, it creates a hypervisor by using VT-x. This hypervisor is responsible for deploying analysis instruments on the environment. The ISKRA framework runs malware samples and performs analysis by monitoring all their system calls. Therefore, a system call monitoring mechanism is needed, which is implemented by the hypervisor.

[33]See https://github.com/ionescu007/SimpleVisor

[34]See https://rayanfam.com/topics/hypervisor-from-scratch-part-7/

Figure 2.3 Inline hooking overview



The Windows operating system provides no legitimate mechanism to directly monitor system calls. On the contrary, it tries to prevent tools that aim to capture system calls by using technologies such as Kernel Patch Protection (KBB or also known as "PatchGuard"[35]). Monitoring tools locate executable code segments of the system calls in the memory and overwrite arbitrary part of it (using JMP instructions or breakpoints) to redirect execution flow to their own code segments. After they complete the necessary operations to log system calls activities, they return execution flow to the instructions of original system calls. This method is known as inline hooking[36], which circumvents KBB. Figure 2.3 shows an overview of inline hooking. The hypervisor in the ISKRA framework uses the same methodology to deploy system call monitoring.

We previously mention that the EPT technology allows us to track and interfere with memory operations. The hypervisor deployed by ISKRA also configures an EPT entry to create EPT violations on reading and writing operations within the memory region of hooks. The processor transmits EPT violations to hypervisors to allow them for handling those violations[37]. Therefore, the hypervisor can detect any read or write operation to memory regions that contain instructions for inline hooking and hide them from other system components such as PatchGuard and malware.

Finally, to log system call executions, the hypervisor adds breakpoints at the memory regions of the system calls, which create violations that are handled by the hypervisor. Therefore, the hypervisor can now log system call executions. The only

[35]See https://bit.ly/3i2OuR7

[36]See https://blog.nettitude.com/uk/windows-inline-function-hooking

[37]For more information about implementation see https://github.com/tandasat/DdiMon

difference of this logging method from a typical inline hook is that it uses breakpoints instead of JMP instructions. As hook handlers are invisible to the guest system, the hook handler would not execute if JMP instructions were used. Therefore, we use breakpoints to redirect execution flow to the hook handler in the hypervisor by triggering violations.

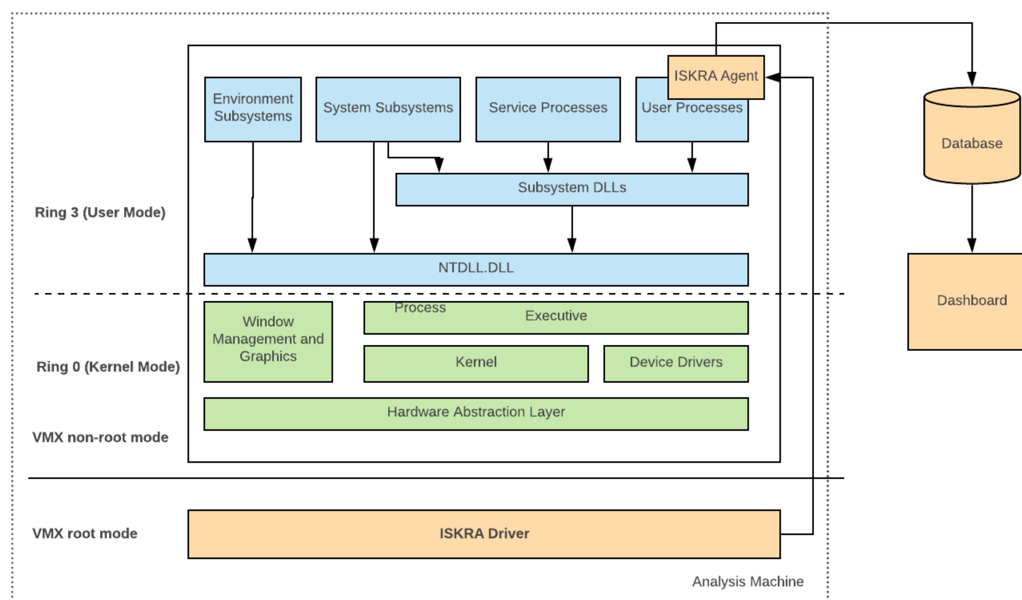
In summary, with the help of the VT-x, we can deploy our framework on a running bare-metal system. It has extensive control over the operating system and applications including malware. Furthermore, EPT allows the privileged framework to interfere with memory operations and hide its presence.

3. ISKRA FRAMEWORK

The emergence and rise of the threat intelligence community has led to relatively quick and prompt discovery of malware attacks at the onset. With public services such as VirusTotal, anyone can easily check if a suspicious software is malicious provided that it is previously analyzed. This shifts the priority of malware developers from having fully undetectable malware to developing malware that can hide its behavior even after it is detected. When malware detects that it is analyzed, it stops running or does not execute its actual malicious code. This prevents the analyst from conducting an in-depth examination of malware, which is essential to perform preventive and remedial actions.

To address this fundamental concern, we present a dynamic analysis framework for use in detailed examination of evasive malware that hides their behavior during analysis. The framework can also be profitably used incident response cases, where the aim is not only detect cyber attacks but also investigate and resolve them. The framework provides analysis in a bare-metal system so that it can successfully detect malicious software that has managed to escape emulation and virtualization based analysis tools. It also allows the incident response operations to be performed directly in the system under attack and malware cannot possibly disrupts the analysis as it cannot detect it. The framework performs malware analysis in three basic stages: i) data collection, ii) decision making, and iii) threat report generation. In the following section, detailed information will be given about these three stages. Figure 3.1 illustrates an overview of the dynamic analysis platform. The figure shows that the ISKRA driver executing in the VMX root mode, which is deployed in a very low level, is responsible to collect data and sends to the ISKRA agent for further process. The driver is in the regular format of a Windows kernel-mode software driver. It is compiled, debugged, and installed as a regular driver executable file.

Figure 3.1 Overview of ISKRA



3.1 Data Collection

In this stage, three types of data are collected to assist malware analysis process:

- System calls
- File header information
- Network packet capture
- Performance counters statistics before and after running malware

The ISKRA framework checks if the monitored system calls reveal any data *pertaining to* the deployment of the analyst tools (such as debuggers, disassemblers, etc.). If that is the case, the framework alter or remove the data in return value, so the presence of analyst tools is hidden from other software in the system such as malware. For example, analysts often use tools such as debuggers and disassemblers, as well as command interpreter (such as cmd.exe) or scripting tools such as Powershell and Python to interact with the system under examination. Since attackers are familiar with common analysis techniques and tools, they check the presence of the tools that are most frequently used in malware analysis. One of the most frequent methods used by evasive malware for this purpose is to obtain a list of running processes in the system and compare it with the list of frequently used

analysis tools. If they detect that an analysis tool is running on the target system, they adapt their behavior in order to hide their malicious intentions.

As a countermeasure to this evasion method, the ISKRA framework controls tool names in the return value of the NtQuerySystemInformation system call, which can be used for obtaining a list of the running processes' names. If the framework detects such malware analysis tools' name in the return value, it hides their presence by removing them from the list. The framework maintains an easily configurable list, therefore the analyst can modify it to hide any tools that is used in the analysis. Thus, the framework can hide both its own components and other malware analysis tools from malicious software.

The ISKRA agent component also extracts file header information, captures network packets and sends them to the database component along with the system calls collected by the driver component. It is possible to use this data, which is obtained and shared in order to provide more detailed information to analysts in the classification of malware with appropriate algorithms (Schultz, Eskin, Zadok & Stolfo, 2001)(Stakhanova, Couture & Ghorbani, 2011).

In a typical malware analysis environment, when the analysis of a malware is completed, the environment is reset to its initial state for the next analysis. So, the tangling of evidence between the two successive analyses in the same environment is avoided. As our framework is designed to be deployed as an automatic dynamic analysis tool, its agent component is responsible for not only resetting, but also recording all system changes during the analysis. Resetting the environment by deleting all changes is a configurable feature of the ISKRA. Thus, the deletion of possible evidence (such as files and registry keys created by the malware) is prevented when the framework is used during incident response. At the end of a predefined automatic analysis period, the ISKRA agent transmits the logs to the database as can be seen in Figure 3.1.

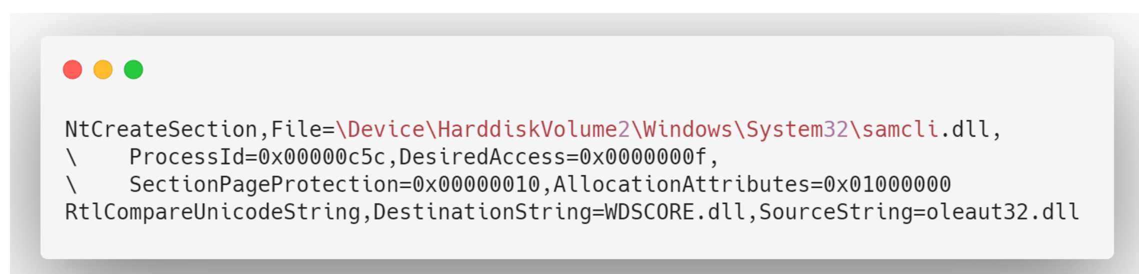
3.2 Data Preprocessing

It this stage, data about malware's dynamic behaviour transmitted by the agent component is parsed and preprocessed; as a result a file containing system calls and its parameters by suspicious executables is created. Example loglines are included in Figure 3.2.

Figure 3.2 has two different calls that are collected from "NtCreateSection" and "RtlCompareUnicodeString" system calls. "NtCreateSection" creates a shared object, which represents a section of memory that can be shared with other processes. Shared objects are also used to map a file into memory address space. In our example, the “AllocationAttributes” parameter has SEC_IMAGE bitmask value (specified in Microsoft Developer Network Documentation), which indicates that this system call is used for mapping an executable file into memory. We can link this information with ProcessId and the file parameters. Line 1 in Figure 3.2 points out that the process with ID 0x00000c5c maps "Device\HarddiskVolume2\Windows\System32\samcli.dll" into its memory. "samcli.dll is a "dynamic-link library (DLL) file, which handles Security Accounts Manager (SAM) operations, through which we understand that a suspicious executable that generates logs in Figure 3.2 intends to interact with users' passwords in the target machine. As illustrated here, an analyst can use this information to understand the malware's capabilities and intentions.

In the second line of Figure 3.2, "RtlCompareUnicodeString" is used to compare "WDSCORE.dll" and "oleaut32.dll" strings. Similarly malware's intentions can be inferred from the logs of "RtlCompareUnicodeString". For instance, evasive malware generally has a blacklist for known analyst tool names such as debuggers, disassemblers, etc.; then it checks the presence of blacklisted process names in the current process list running on the target machine. To this end, malware makes string comparisons between the current process list objects and blacklist objects. So if an analyst sees malware analyst tools names in string comparison in "RtlCompareUnicodeString" logs, the analyst deduces that evasive malware checks the presence of analyst tools in the target machine.

Figure 3.2 Captured System Call Example



```
NtCreateSection,File=\Device\HarddiskVolume2\Windows\System32\samcli.dll,
\    ProcessId=0x00000c5c,DesiredAccess=0x0000000f,
\    SectionPageProtection=0x00000010,AllocationAttributes=0x01000000
RtlCompareUnicodeString,DestinationString=WDSCORE.dll,SourceString=oleaut32.dll
```

3.3 Threat Report Generation

It this stage, a threat report is generated that can be examined by the analyst. This report, which is created in JSON format for easy manual editing or interpretation through software, contains essentially the following information:

- Main Portable Executable (PE) header information. This header contains all necessary information for the Windows operating system to load an executable into memory and execute it. An analyst uses that information to verify that the current environment meets the executable's requirements. For example, if the PE header specifies the .NET framework requirement, an analyst understands that the framework must be installed in the environment to execute the malware.
- Process list
- System calls (together with their parameters) made by the suspicious process
- Summary of HTTP and DNS requests extracted from network packet capture.

The threat report obtained is recorded in the database and presented to the analyst through a user-friendly web interface.

4. BEHAVIOUR DATA MONITORING

In this section, we provide detailed information about the dynamic analysis framework component that not only detects evasive malware, but also monitors its dynamic behavior.

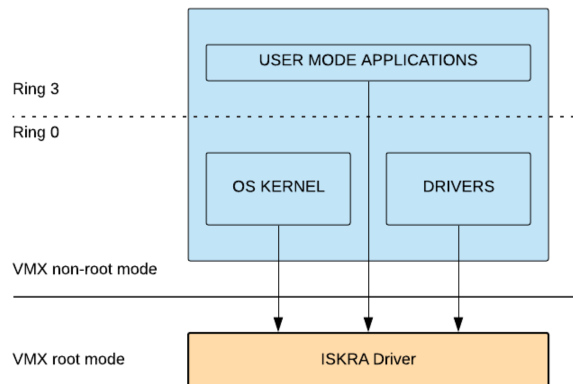
The main method used by evasive malware, which is the subject of our study, to avoid analysis by security products, is to profile its execution environment and to refrain from performing its harmful activities when it detects an environment, which is dissimilar to typical user profile such as emulation tools. For this reason, analysts want to obtain the most detailed information about the behavior of malware while leaving minimal traces in the system. However, emulation and traditional virtualization technologies used today can be easily detected by malicious software due to their physical differences from the bare metal system (Yokoyama, Ishii, Tanabe, Papa, Yoshioka, Matsumoto, Kasama, Inoue, Brengel, Backes & Rossow, 2016). Moreover, these methods, which are based on the external intervention of the entire operating system, become unusable when the analyst wants to move to a bare metal system.

The dynamic analysis framework proposed in this study employs the Extended Page Tables (EPT) feature provided by Intel VT-x technology[1] to enable behavior monitoring without being detected by malware. With the VT-x technology that comes with the second-generation Intel processors, virtual machine monitoring has become a much more useful method. It enables the development of faster, less costly, and more effective virtualization solutions. The technology brought the concepts of the VMX root mode and the VMX non-root mode in addition to the privilege levels commonly called Ring 0 through Ring 3. While the root mode is developed for use by the host system, the non-root mode is developed for use by the virtual machine. With the EPT feature, which was later added to this technology, it is possible to virtualize the memory management unit (MMU) processes, so that a bare metal hypervisor running in the VMX root mode can have full control over the code and data in the

[1] See https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html

virtual machine. As of today, Intel VT-x and EPT technologies allow monitoring

Figure 4.1 Overview of kernel component



from a level that has more privilege than malware and low level software including the operating system and rootkits. Our dynamic analysis framework monitors system calls using these advantages brought by Intel VT-x and EPT technologies. It simply monitors system calls with the HyperPlatform hypervisor (Korkin & Tanda, 2017). Figure 4.1 shows an overview of the driver component and how it is situated with respect to OS kernel and user mode applications.

5. MODEL CONSTRUCTION

In our experiments, system call sequences are examined using different approaches with different machine learning and deep learning algorithms. In this section, we explain approaches and methodologies used in our experiments. Then, the subsequent section will present experimental results.

In one approach, machine learning models are tested using TF-IDF values of n-grams of system calls. After the pre-processing step, a separate file is created in the database containing system calls made by a specific malware sample. TF-IDF values of n-grams are extracted from these files and used as features in the machine learning models. During the tests, separate models are created and evaluated with different n-gram lengths and machine learning algorithms. For n-gram notations in this study, we use single length n-grams (such as $n = 1$ for monograms) and n-gram ranges (such as $n = [2,4]$). The notation, where the range is used, means that n-grams are selected for all lengths within that range. For example, $n = [1,2]$ means monograms and bigrams are used.

In another approach, the system calls are converted to number sequences, and modeling and evaluation are performed by applying the Long Short Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) algorithm on these sequences. A unique integer value is assigned for each different system call for use in the data processing phase, so that system call logs are converted to number sequences. Table 5.1 shows an example subset of the mapping between system calls and corresponding unique integers.

In both approaches, instead of treating them separately, we group system calls that

RtlInitUnicodeString	1
NtSetInformationFile	2
NtCreateSection	3
NtAllocateVirtualMemory	4
NtCreateFile	5

Table 5.1 An Example Subset Of The System Calls And Their Unique Integer Values

Systam Call Category	System Calls
Memory	MmCopyVirtualMemory, NtCreateSection NtAllocateVirtualMemory, RtlDecompressBuffer, RtlDecompressBufferEx
File	NtCreateFile, NtDeleteFile, NtQueryDirectoryFile, NtReadFile, NtSetInformationFile, NtDeviceIoControlFile, NtWriteFile
Process	NtOpenProcess, NtCreateProcess, NtOpenProcessToken, NtOpenThreadToken, PsCreateSystemThread, PsSuspendProcess, PsResumeProcess
String	RtlAnsiStringToUnicodeString, RtlUnicodeStringToAnsiString, RtlInitUnicodeString, RtlCompareUnicodeString RtlInitAnsiString
Privilege	NtAdjustPrivilegesToken, NtDuplicateToken SeAccessCheck,
Registry	RtlCreateRegistryKey, RtlQueryRegistryValuesEx, RtlDeleteRegistryValue, RtlRtlCheckRegistryKey

Table 5.2 Hooked system calls and their corresponding categories

operate with similar operating system objects into categories and report the effects of grouping on the accuracy of the machine learning models. Table 5.2 shows system call categories.

In yet another approach, we use system call parameters as inputs in our machine learning algorithms and show that they can be used to distinguish the malicious behavior with high accuracy.

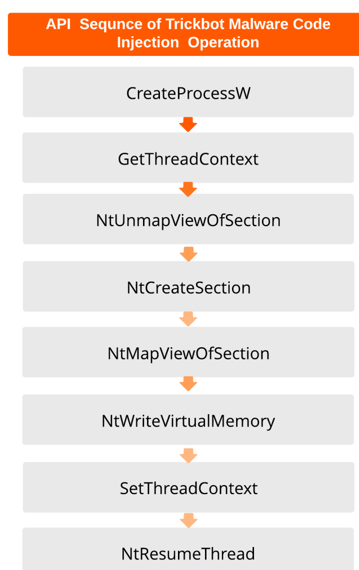
All software, including malicious software, interact with the operating system to fulfill their fundamental functionality. User mode software often uses the application programming interface (API) provided by the operating system when interacting with the system and other system components in order to ensure compatibility with frequently released operating system updates and to reduce the executable size. When these APIs will perform low-level operations that are handled by the kernel of the operating system, such as hardware-related operations or creating new processes, they make the relevant system calls and turn over the execution of the related task to the kernel.

Unlike API calls, many system calls are not made available directly to software developers in the Windows (and many other) operating system, and not all system calls provided by the operating system are documented. However, thanks to endeavors of the reverse engineering community, it is now elementary for any software to use these system calls directly. To summarize, software that performs low-level operations gives rise to system calls either through user-mode APIs or directly through the kernel.

Malware is known to attempt to evade security solutions by calling system functions directly instead of user mode APIs. For instance, as many security products monitor user mode APIs, malware can easily bypass them by interacting with kernel directly.

Also, since system calls operate on low-level system components, it is a laborious

Figure 5.1 Example API Sequence for Code Injection



endeavor to re-implement them. After all, any error in their implementation may cause critical system failures. To avoid this, malware developers, therefore, tend to invoke available system calls instead of re-implementing them as do other legitimate programs. Consequently, by monitoring system calls we can observe and investigate interactions of malicious software with the operating system reliably to a greater extent.

A single system call performs one primary operating system job. But even a single task that the malware wants to perform is more involved and often leads to multiple system calls. For example, a single malicious task such file stealing may cause system calls for obtaining file handle (e.g. NtCreateFile) and reading file (e.g. NtReadFile). Therefore, we examine the system calls as system call sequences to capture their intentions and context.

The NtCreateSection system call can be used either by a benign program to map data to memory or by malicious software to run its code in another process by employing what is known as *code injection*. However, if the other system calls are also invoked, then one can infer if the call is a part of a malicious action. Figure 5.1 shows the API sequence generated by malware called Trickbot, which is an infamous financial Trojan family, while performing code injection. Fundamental steps of this operation listed below:

- Create a new suspended process
- Retrieve the context of its main thread

- Allocate sufficient memory space for malicious code at the target process's memory range
- Write its malicious code to the allocated memory space
- Set start of the malicious code as the thread instruction pointer
- Resume the thread

As can be understood from the example, an ordered set of system calls proves to be instrumental to distinguish a malicious act.

Originally, we aim to monitor system call parameters to obtain in-depth information about system calls and eventually create indicators of compromise (IOC) lists from string valued parameters, such as file paths, process names, and registry paths. Also, particular string values of system call parameters prove to be reliable features in our machine learning algorithms to classify malicious behavior. We also report their performance as an alternative and competitive method for malware classification.

Additionally, the ISKRA collects the network traffic, PE file header information, and hardware performance counters (HPC). For this study, however, we don't leverage the network packets and file headers collected by the ISKRA. Similarly, we do not use hardware performance counters-based (HPC-based) sampling within the scope of this paper. However, we facilitate HPC sampling in the framework as they prove to be instrumental in detecting sophisticated attacks such as cache-based attacks (Kulah, Dincer, Yilmaz & Savas, 2019) (Chiappetta, Savas & Yilmaz, 2016).

6. EXPERIMENTAL RESULTS

The proposed dynamic analysis platform is tested using two different methods. In the first method, recent evasive malware samples are identified in the current campaigns using our field experience and verified by manual methods. In addition to the experiments with our framework, we analyzed the malware samples detected by manual methods using other existing dynamic analysis solutions and compare their performance with that of our framework. In the second method, experiments are carried out on samples obtained by random sampling from the malware data sets provided by the suspicious file inspection service, called VirusTotal.

6.1 Real World Examples

This section includes recent examples of malware campaigns observed worldwide in 2020, for which all necessary indicators of compromise (IOC) are not detected by the existing analysis solutions. Recall that the IOC includes the necessary information to detect, analyze and prevent malware operations. Although there are various proposals in the cyber security community for the classification of IOC types, there is no generally accepted IOC type classification scheme. Within the scope of this study, the performance of a dynamic analysis platform in extracting the necessary IOC information is evaluated with its success in finding file system and network traces, which provide the most essential information. This includes file system changes and data such as remote server address, port, and protocol used by malware to communicate with its command and control server, which is the most distinguishing information left by an evasive malware sample. Therefore, it is essential to ensure that malware executes in the analysis environment, fulfills its original intent and reveals these types of information.

In the following, we provide the details of our study with three evasive malware

samples that cannot be detected by existing dynamic analysis tools, which include

- ANY.RUN[1]
- JOE Sandbox[2]
- CAPE SAndbox[3]

6.1.1 Zloader

Zloader malware, a derivative of the Zeus malware, has been distributed since February 2016. Cybercriminals launched worldwide Zloader attacks targeting users during the coronavirus outbreak in 2020. Zloader can be used to install new malware on victim machines and can also steal victims' passwords using "the man-in-the-browser" method. In the method, Zloader injects malicious code into running web browser processes in the victim machine and then hooks network relevant functions invoked within the browser process so that malware can sniff usernames and passwords.

For attackers, it is not a straightforward undertaking to convince potential victims to execute an arbitrary executable file in an e-mail attachment. Therefore, in the Zloader malware campaign, attackers carry out their operation in two stages. In the first stage, spam e-mails with malicious Microsoft Office Excel attachments are sent to potential victims. These spams contain phishing text to trick targets into opening the file in the e-mail attachment. These Excel files contain malicious macro codes, which are written in VBA (Visual Basic for Applications) language, to download and execute actual Zloader malware payload. The second stage starts when the malicious code is executed.

This attack campaign employs effective methods to evade automated dynamic analysis platforms. As a result, the existing dynamic analysis services cannot completely analyze and collect all necessary information to profile the malware behaviour since it cannot be monitored dynamically even when it is found to be malicious via signature-based methods. When the evasion methods applied by the Zloader campaign are examined with reverse engineering tools and techniques, it can be seen that malicious macro codes in the first stage of the attack collect information about the working

---

[1] See https://any.run/

[2] See https://www.joesandbox.com/

[3] See https://capesandbox.com/

environment, and based on these information they can detect automatic analysis solutions. More specifically, these malicious macro codes, which execute in the process context of the Excel process, obtain information about window GUI properties of the Excel process, macro debugger presence, and operating system properties. Existing dynamic analysis platforms generally use command-line tools to easily automate the analysis pipeline and use low display resolutions to reduce virtual machine system requirements. The malicious macro codes use window properties of the Excel software user interface and compare them with widely used end-user system properties. Based on this comparison, malware can differentiate anomalous systems such as those deploying dynamic analysis environments. In addition, many dynamic analysis solutions implement only necessary parts of the actual physical computer to operate at a minimum cost. The malicious macro codes, then, check the presence of mouse and audio system in the working environment to catch the presence of dynamic analysis platforms. The following list shows all information used by macro in the Zloader malware[4] to this end:

- Is the window hidden?
- Is the window maximized?
- Window size
- Is the malicious macro code debugged?
- The workspace width
- the workspace height
- Whether a mouse is plugged
- Whether the computer can play audio

Figure 6.1 Zloader Network Traffic Captured by the ISKRA

```
0x4c74 A japanjisho.info A 104.24.105.178 A 104.24.104.178
gavrelets.ru
0x0ff8 A gavrelets.ru A 92.53.96.168
```

To evaluate the effectiveness of current solutions for Zloader, we searched public analysis reports of it in three different online analysis platforms. These platforms are Joe Sandbox, ANY.RUN and CAPE Sandbox. We observed that even if these platforms successfully classify malware samples by matching signature, they could not reveal the malware's command control server because they were detected by

[4] See https://www.lastline.com/labsblog/evolution-of-excel-4-0-macro-weaponization/

the evasion methods[5,6,7]. Figure 6.1 shows a network packet capture of the Zloader command and control server address resolution obtained by ISKRA.

6.1.2 GuLoader

GuLoader is a VB5/6 downloader that was active in the first half of 2020. Unlike the Zloader malware, it has no harmful functions beyond downloading other malware samples (generally those intended for online banking users) in the target system. We observe that the existing analysis solutions are again insufficient due to the anti-analysis methods employed by this campaign. GuLoader efficiently uses well-known anti-analysis methods to evade current solutions, which are explained below:

Virtual machine related strings It checks for the presence of "C:\ProgramData\qemu-ga\qga.state" file in execution environment. This file is a part of QEMU Guest Agent software[8].

Number of application windows It counts the number of top-level application windows by using EnumWindows Windows API. If the number is less than 12, then the malware terminates itself[9].

To evaluate the effectiveness of current solutions for Guloader, we obtained a sample Guloader[10] by our private sector partnership and analysed it with three different online analysis platforms. Our observations are explained in the following.

Joe Sandbox It successfully classified the sample as GuLoader; but the report clearly states that the detection depends on behaviour signature. There is no network traffic in the report. This shows that the malware abstains from fulfilling its fundamental function and downloads no other malware. Network traffic

[5]https://capesandbox.com/analysis/52317/

[6]https://app.any.run/tasks/5361f747-19ac-4877-bde0-fb0cc86ecd3c/

[7]https://www.joesandbox.com/analysis/257133/0/htmlnetwork

[8]See https://blog.vincss.net/2020/05/re014-guloader-antivm-techniques.html

[9]https://www.crowdstrike.com/blog/guloader-malware-analysis/

[10]SHA1: 2ee96ccec4d361aebe8540492f233491b386caa7

information is especially important to learn more about the attack campaign and its origins; but the Joe Sandbox failed to provide it.

ANY.RUN It failed to classify the GuLoader sample as malware. There is no detected malware artifact, network or file operation in the report.

CAPE Sandbox It only detects a static signature of the sample. There is no implication in the report that points to the download and execution of a malware sample.

In our experiment on the relevant harmful executable, ISKRA framework successfully classified the sample as malicious by using its machine learning based classifier. Also, the ISKRA captured network traffic between the malware and its command and control server. Figure 6.2 shows the HTTP traffic summary generated by the ISKRA framework. An analyst that can access this summary from the framework dashboard, easily obtains necessary network IOC to detect and block the GuLoader campaign without further manual investigation on the malicious executable.

Figure 6.2 GuLoader Network Traffic Captured by the ISKRA

	Command & Control Server Address	HTTP URL for Command & Control Server Endpoint
192.168.1.108	185.230.160.192	185.230.160.192/test/index.php
192.168.1.104	239.255.255.250	239.255.255.250:1900 *
192.168.1.108	185.230.160.192	185.230.160.192/test/index.php
192.168.1.109	239.255.255.250	239.255.255.250:1900 *

6.1.3 MassLogger

MassLogger is a stealer developed using the .NET programming language employed by attacks in 2020. This malware, which steals user passwords and sends this data over SMTP, is another family of malware that bypasses current dynamic analysis platforms. It detects analysis platforms by using various methods as explained in the following.

Virtualization Software Detection MassLogger initially gets computer

system properties by querying Windows Management Instrumentation (WMI) with “Select * from Win32_ComputerSystem". Then, it compares computer manufacturer property with widely used virtualization software names. If the computer manufacturer’s name is Vmware or Virtualbox, the malware terminates itself[11].

Analysis Component Detection MassLogger checks widely used sandbox components in the execution environment. For instance, to detect Sandboxie open source sandbox, the malware checks existence of "SbieDLL.dll" module, which is the Sandboxie sandbox’s user-mode component, by invoking GetModuleHandle Windows API.

By our private sector alliance, we obtained a Masslogger malware sample[12]. Then, in order to examine the analysis results of current solutions, we uploaded the sample into the three dynamic analysis platform and obtained their reports, which are summarized in the following.

Joe Sandbox The report states that this sample is classified by a static signature match on unpacked binary[13]. Reverse engineering on the malware sample proves that Masslogger sends stolen data over SMTP. But the Joe Sandbox report has no SMTP related network traffic. Thus, the report returned by Joe Sandbox is missing network-related IOC information.

ANY.RUN The report[14] returns “No threats detected” as a result and has no data indicating harmful operations of the MassLogger sample.

CAPE Sandbox It classifies the sample as Masslogger. But the report[15] declares that the sample is firstly processed by the “Unpacker" task to decompress it. After that, the sample is classified by scanning static signatures on the processed binary file. However, these two steps are essentially based on the previous manual analysis of the relevant malware family. Therefore, they need to be updated as the malware sample is updated. Command and control server hostname is shown in the “DNS" section of the CAPE Sandbox report. But there is no SMTP traffic

[11]See https://fr3d.hk/blog/masslogger-frankenstein-s-creation

[12]SHA1: 851c8818d44587d188dde10cee1eda582e97118e

[13]See https://www.joesandbox.com/analysis/237655/0/html

[14]See https://app.any.run/tasks/a48223e2-fea3-44ac-817e-3a976ae2f9a5/

[15]See https://capesandbox.com/analysis/42194/

captured. Thus, an analyst with no preliminary information about the Masslogger, cannot relate DNS resolution with any other information in the report. Another consequence of the lack of SMTP traffic in the CAPE sandbox report is that stolen data from the target system cannot be observed.

Figure 6.3 MassLogger Network Traffic Captured by ISKRA

```
MIME-Version: 1.0
From:
To:
Date: 11 Jun 2020 04:54:17 -0700
Subject: MassLogger | @United States
Content-Type: multipart/mixed;
 boundary=--boundary_

----boundary_
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: quoted-printable

zip attachment
----boundary_
Content-Type: application/octet-stream; name="_United
 States .zip"
Content-Transfer-Encoding: base64
Content-Disposition: attachment

UEsDBBQAAAAAAMUmy1AAAAAAAAAAAAAAAAA0ACQAbWFsd2FyZV9Vbml0ZWQgU3Rhd
```

In our experiment for the same sample on the ISKRA dynamic analysis framework, we successfully obtained network packet capture between the malware and its command and control server. Figure 6.3 shows the details of harmful SMTP traffic (some parts redacted for concealing stolen data). In summary, the ISKRA framework provides below information with these traffic:

- Command and control SMTP server address from "To" field.
- Malware family name from "Subject" field.
- Stolen infomation from victim device. Because stolen data is sent in plain over SMTP.

An analyst can easily extract ZIP attachment from the network packet capture and then reveal malware configuration, which can be seen in the Figure 6.4.

6.1.4 Cradle Ransomware

Cradle is a ransomware family that was disclosed by security researchers in 2017[16]. This ransomware is sold as ready to use a crimeware kit in Dark Web. Also, an

[16] See https://www.cyber.nj.gov/threat-center/threat-profiles/ransomware-variants/cradlecore

Figure 6.4 MassLogger Configuration Extracted from the ISKRA logs.

```
#################################################################
MassLogger v1.3.6.0
#################################################################

### Logger Details ###
User Name:
IP:
Location: United States
Windows OS: Microsoft Windows 10 Home64bit
Windows Serial Key:
CPU: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
GPU: Intel(R) UHD Graphics 620
AV: Windows Defender
Screen Resolution: 1366x768
Current Time: 6/11/2020 4:54:09 AM
MassLogger Started: 6/11/2020 4:54:07 AM
Interval: 1 hour
MassLogger Process: C:\Windows\Microsoft.NET\Framework\v4.0.30319\MSBuild.exe
MassLogger Melt: false
MassLogger Exit after delivery: false
As Administrator: False
Processes:
Name:
Name:

### WD Exclusion ###
Disabled

### USB Spread ###
Disabled

### Binder ###
Disabled

### Window Searcher ###
Disabled

### Downloader ###
Disabled

### Bot Killer ###
Disabled

### Search And Upload ###
Disabled
```

attacker can buy this malware as a source code project to make her/his own modifications. It targets more than 300 file types in Windows operating system and encrypts them with the Blowfish blockcipher encryption algorithm. A cybersecurity organization found an advertisement for Cradle ransomware in an underground forum[17]. The advertisement reveals that the ransomware has an evasion feature. A simple, but effective feature is based on the CPU core number of the target machine. Modern personal computers usually have a multicore CPU. However, some dynamic malware analysis platforms use single-core systems while building an analysis environment to limit the system cost. Thus, attackers use this dissimilarity to detect dynamic analysis platforms as in the case of Cradle.

To assess the effectiveness of the current platforms for Cradle, we obtained a sample of this malware campaign[18] by our private sector partnership and analyzed it with different online analysis platforms. Our experiments are explained in the following.

---

[17] See https://www.forcepoint.com/blog/x-labs/cradlecore-ransomware-source-code-sale

[18] SHA1: a2a164a4a535c5542accb45d1268ac072b48ff1a

Joe Sandbox The Joe sandbox report[19] for the sample contains network IOC information. But the "File Activities" section in the report has no indicator for file operations of the ransomware sample. File IOC is the most essential information for detecting and analyzing ransomware families as they rename encrypted files with a unique extension. This suggests that Joe Sandbox fails to execute the sample.

ANY.RUN The report[20] has no IOC for file or network operations of the ransomware. Apparently, ANY.RUN fails to execute the sample.

CAPE Sandbox CAPE Sandbox is the only public solution among those we tested that is able to run Cradle. The report contains the command and control server address and lists renamed encrypted files.

In our experiments, the ISKRA framework manages to obtain all necessary network and file indicators. Detected indicators contain command and control server address and modified file list. Figure 6.5 shows the ransom note created by Cradle ransomware. This ransom note is captured by the ISKRA framework from file modification logs. Hence, an analyzer easily identifies malware family and its purpose by using this ransom note.

Figure 6.5 Ransom Note Obtained with the ISKRA.

!!! ATTENTION !!!

What happened to my files?

pictures, music, and other important files are locked. Your files are locked with military-grade encryption

It is impossible to unlock your files without the secret password and decryption software.
Do not try to unlock the files yourself -- you will certainly damage your files if you try.

6.2 Samples from VirusTotal

To evaluate the performance of our dynamic analysis platform in monitoring malware and the usability of the data collected by this platform, we experimented with 72

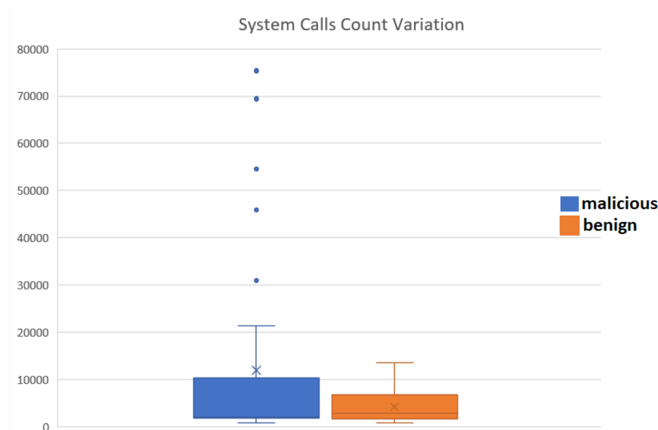
[19]See https://www.joesandbox.com/analysis/267105/0/html

[20]See https://app.any.run/tasks/9fedbfcc-c6d0-4a18-a32a-8a3440211201

sample software files. While 32 of the files are benign 40 of them are known malicious files, which are obtained by random sampling from the VirusTotal dataset. Each malicious sample and its report, which is created by the ISKRA framework, are manually analyzed and it is observed that the actual purpose and behavior of the malware are identical. This shows us that our dynamic analysis platform can run the malware sample and collect dynamic behaviour data without being detected by current malware campaigns.

For the benign sample collection, we crawled files from two different sources. The first source is a free portable software website[21]. For this, we developed a web crawler and downloaded 100 portable software samples. The second source for benign files is an end-user machine. We wrote a file crawler for the Windows operating system and collected 792 executables from a freshly installed system. After collecting all these samples, we selected 32 benign files by random sampling. Each test sample is executed for 5 minutes in the test environment. 80% of the sample set are reserved for training, the other 20% for testing. In the experiments, a total of 628,463 system calls are collected by executing 72 samples. A total of 501,522 of these system calls are collected from the malicious samples. The average number of system calls collected from a malicious sample is 11,941 for the benign samples, with an overall average of 8411. Figure 6.6 summarizes the system call counts of the captured logs.

Figure 6.6 System Calls Count Variation



In the experiments, we tried different machine learning classification methods and reported their performance (cf. Tables 6.1, 6.2, 6.3, 6.4). As the signature-based detection is not used and a malware sample is used in the classification phase for the first time, it is considered as *zero-day* malware.

We first worked with mono-grams (i.e., $n = 1$) of raw system call sequences and calculated their TF-IDF values. Two cases are considered. In Case I, (See Table 6.1),

[21]https://www.portablefreeware.com/

Classifier	Precision	Recall	AUC
LogisticRegression solver='lbfgs'	0.63	0.83	0.57
SVC kernel='rbf'	0.58	1.00	0.50
SVC kernel='linear'	0.65	0.79	0.59
DecisionTreeClassifier, max_depth=3	0.75	0.64	0.67
DecisionTreeClassifier, max_depth=4	0.61	0.60	0.53
KNeighborsClassifier, k=2	0.74	0.33	0.58
KNeighborsClassifier, k=3	0.67	0.43	0.56
KNeighborsClassifier, k=4	0.82	0.33	0.62

Table 6.1 Case I: classification results for all system calls using monograms (i.e., $n = 1$) and dictionary size is 22

Classifier	Precision	Recall	AUC
LogisticRegression solver='lbfgs'	0.65	0.93	0.61
SVC kernel='linear'	0.58	1.00	0.50
DecisionTreeClassifier max_depth=3	0.73	0.83	0.70
DecisionTreeClassifier max_depth=4	0.75	0.79	0.71
KNeighborsClassifier, k=2	0.84	0.62	0.73
KNeighborsClassifier, k=3	0.71	0.69	0.65
KNeighborsClassifier, k=4	0.81	0.62	0.71

Table 6.2 Case II: classification results for system calls in the sample data set using monograms (i.e., $n = 1$) and dictionary size is 21

in addition to those in the data set, system calls from all processes that happen to be running are used for the training and classification phases. This case represents a noisy setting. In Case II (See Table 6.2), only the system calls in the sample data set are used. Case II represents a typical analysis scenario, in which the analyst inspects a given sample. Table 6.1 shows that the best AUC is %67 for Case I and it is obtained with the Decision Tree classifier. Also, Table 6.2 shows that the best AUC is %73 for Case II and it is obtained with the k-nearest neighbors classifier. When we compare the best AUC scores of Case I and Case II, we observe that the best AUC in Case II is 6% better than the one in Case I.

As expected, Case I, which is noisy, is less successful than Case II. All the same, the AUC score of 67% (See Table 6.1) shows that Case I is still highly valuable for analysis by reducing the number of suspicious applications in cases, where the source of the incident is unknown such as digital forensic and incident response applications. Note that in those types of applications we envisage that forensic experts or incident response team members work with a running system and are not provided with samples, but inspect calls from all types of processes to find out the culprit.

Classifier	**Precision**	**Recall**	**AUC**
LogisticRegression solver='lbfgs'	0.67	0.69	0.61
SVC kernel='linear'	0.76	0.60	0.66
DecisionTreeClassifier max_depth=3	0.78	0.74	0.72
DecisionTreeClassifier max_depth=3	0.75	0.64	0.67
KNeighborsClassifier, k=2	0.88	0.33	0.63
KNeighborsClassifier, k=3	0.76	0.45	0.63
KNeighborsClassifier, k=4	0.93	0.33	0.65
Randomforest, n_estimators=5	0.64	0.69	0.58
Randomforest, n_estimators=10	0.70	0.67	0.63

Table 6.3 Case I: results for n-gram range of [2, 4] and dictionary size is 3725

Classifier	**Precision**	**Recall**	**AUC**
LogisticRegression solver='lbfgs'	0.66	0.83	0.62
SVC kernel='linear'	0.68	0.71	0.62
DecisionTreeClassifier max_depth=3	0.86	0.86	0.83
DecisionTreeClassifier max_depth=4	0.81	0.81	0.77
KNeighborsClassifier, k=2	0.76	0.67	0.68
KNeighborsClassifier, k=3	0.67	0.83	0.63
KNeighborsClassifier, k=4	0.76	0.69	0.70
Randomforest, n_estimators=5	0.76	0.90	0.75
Randomforest, n_estimators=10	0.82	0.88	0.81

Table 6.4 Case II: results for n-gram range of [2, 4] and dictionary size is 3899

Next, we investigated the effect of n-grams of system calls by experimenting with various ranges of n. With the help of the `TfidfVectorizer` function in the scikit-learn library, TF-IDF values of n-grams of system calls are extracted in various ranges of n and experiments are carried out with machine learning algorithms. The algorithms we tested during the experiments are Logistic Regression, Support Vector Classification, Decision Tree, k-nearest neighbors algorithm (k Neighbors), Random Forest. We chose these algorithms as we observed that they are frequently used in malware classification tasks. The highest success rates in the experiments are obtained in the n-gram range of [2, 4]. Table 6.3 shows the results of the experiments for system calls collected from all processes (Case I), and Table 6.4 shows the system calls in the sample data set (Case II). The figures in the tables show that grouping system calls as n-grams improves the detection performance and the AUC can exceed 80%.

After experimenting with classical machine learning algorithms, we investigated the performance of more sophisticated deep learning techniques in developing our decision-making model. Since our data set consists of system call sequences, we opt for the long short-term memory (LSTM) networks due to their success in processing

Parameter	Possible Values
Hidden dimensions	10, 40, 70, 100
Dropout Rate	0.4, 0.5, 0.6
Batch Size	32, 64, 128
Max Epochs	10, 20, 30

Table 6.5 Tested hyperparametes for the LSTM networks

Score Type	Value
Accuracy	0.8333
Precision	0.8333
Recall	0.8333
F1-Score	0.8333
AUC	0.8333

Table 6.6 LSTM experiments results for Case II

data sequences (Vinayakumar, Soman, Poornachandran & Sachin Kumar, 2018). We used the PyTorch machine learning library to implement the experiments. To analyze the system call sequences by using the LSTM networks, we have to work with numeric arrays. With the `sklearn.preprocessing.LabelEncoder` function, we transformed the system call sequences into integer arrays. We divided the system call sequence data set of 72 samples into training and test sets with reserving 20% for test data set size. We utilized `hyperparameter` optimization option with `GridSearchCV` function of the scikit-learn library to create the most suitable LSTM network for our data set. The possible parameter set that are used during parameter optimization tests are given in Table 6.5. The `GridSearchCV` function reports that the most successful parameter combination is as follows: Max Epochs = 10, Dropout Rate = 0.4, Hidden Dimensions = 10, Batch Size = 32.

Using the LSTM network generated with the best hyperparameters, the performance scores given in Table 6.6 are obtained. As can be observed in the table the AUC score can get as high as 83.3%, which is in fact better than those obtained using classical machine learning algorithms (cf. Tables 6.1, 6.2, 6.3, 6.4).

6.2.1 String Parameter Classification

In addition to system calls, we collected their associated parameters. In particular, the string parameters of the system calls collected in the experiments are extracted and analyzed with machine learning. Although all types of system call parameters

Classifier	Precision	Recall	AUC
LogisticRegression solver='lbfgs'	0.55	0.78	0.47
SVC kernel='linear'	0.54	0.71	0.45
DecisionTreeClassifier max_depth=3	0.85	0.80	0.80
KNeighborsClassifier, k=3	0.6	0.5	0.52
RandomForestClassifier (n_estimator = 5)	0.67	0.73	0.62

Table 6.7 Classification scores of string system call parameters using various machine learning algorithms (Case II and $n = 3$)

Classifier	Precision	Recall	AUC
(2, 3)	0.88	0.86	0.85
(2, 4)	0.88	0.83	0.84
(2, 5)	0.90	0.88	0.88
(2, 6)	0.92	0.83	0.87
(2, 7)	0.89	0.79	0.83
(2, 8)	0.89	0.79	0.83
(2, 9)	0.86	0.88	0.84
(2, 10)	0.92	0.81	0.86

Table 6.8 String classification using decision tree algorithm (Case II and $n \in [2,10]$)

(such as handles and kernel-mode data structures) can be utilized, our field experience on this subject suggests that string type parameters provide more insight about malware with less preprocess steps on the logs. Thus, only the string type parameters are used during the experiments in this section.

In the experiments, a total of 186723 strings are collected by executing 72 samples. A total of 118,739 these strings are collected from the malicious samples. For feature extracting and model training for string parameter classification, we used TF-IDF values of n-grams of strings. It is a similar method that we used in the previous section for system call sequences classification. Table 6.7 lists the detection scores obtained using various machine learning algorithms for tri-grams ($n = 3$). The dictionary size of tri-grams ($n = 3$) is 161,205.

As can be observed in Table 6.7, the best method turns out to be the decision tree algorithm with an AUC of 80%. Therefore, we further our experiments with decision tree using n-grams for different ranges of n and show the resulting performance in Table 6.8. As can be seen in the table, n-grams of system call string parameters for $n \in \{2,3,4\}$, which have 479,684 dictionary size, result in an AUC score of as high as 88% for a decision tree of maximum depth of 5, which is the highest value achieved in all our experiments.

System Specification	
Computer	Lenovo ThinkPad E480
OS	Windows 10 Home
OS Build Version	16299
CPU	Intel Core i5-8250U
Memory	4 GB

Table 6.9 Test Environment System Specification

Score	Before Monitoring	While Monitoring
CPU Score	899	896
Float Ops	196996727	196451877
Integer Ops	1276258232	1268521848
Hash Ops	1393523	1392959
Ram Score	135	134
RAM Speed	11556 MB/s	11418 MB/s
Disk Score	34	34
Write Speed	138 MB/s	129 MB/s
Read Speed	133 MB/s	136 MB/s

Table 6.10 Benchmark Results

6.3 Benchmark

In this section, we report the overhead in system performance scores when we run our dynamic analysis framework in a notebook computer, whose specifications are listed in Table 6.9.

Using the benchmark software Novabench (see `https://novabench.com`), performance tests are carried out before and after running our dynamic analysis platform in the physical environment in Table 6.9. Performance metrics such as "CPU Score", "Float Ops" are listed in the first column of Table 6.10. The results given in the Table 6.10 represent the two benchmark measures made during the experiments. The "Before Monitoring" column shows the measurements results before running ISKRA. The "While Monitoring" column shows the measurement results while ISKRA running and collecting system call logs. "CPU Score" in Table 6.10 shows the evaluation result for CPU score. The "Ops" values in the following rows represent the number of instructions in different sets (such as Float and Integer) that are executed during evaluation. "Ram Score" in Table 6.10 shows the evaluation result for memory transfer performance. "Disk Score" in Table 6.10 shows the evaluation result for direct, sequential disk read and write speeds. The benchmark scores in the table suggest that the overhead in overall system performance is negligibly low. This ba-

sically means that the computer system that is turned into an analysis environment can continue functioning without significant adverse effect on its performance. The low performance overhead is especially important in incident response cases as the proposed analysis platform can run even in computers with low performance.

In addition, the change in the system performance cannot be distinguished from normal fluctuations in performance. Consequently, we think that the overhead is not enough for malware to detect the presence of ISKRA by side-channel analysis.

7. RELATED WORK

As malware is probably the most essential instrument utilized by cyber attackers, malware analysis is the primary course of action taken by security analysis teams to battle cyber attacks. However, the increasing number and sophistication of malware attacks and their involved evasion techniques render the task of security analysts difficult. Therefore, stealthy, accurate, and fast dynamic malware analysis is an open problem. This section summarizes the solutions suggested in related works in the literature.

Although solutions based on virtualization Cuckoo Sandbox[1] and emulation Anubis[2] provide analysts with straightforward installation and usage, they can be easily detected by hardware fingerprinting methods due to their differences from bare metal physical systems. Drakvuf (Lengyel, Maresca, Payne, Webster, Vogl & Kiayias, 2014), CXPInspector[3], and Ether (Dinaburg, Royal, Sharif & Lee, 2008), which are hypervisor-based dynamic analysis solutions, depend on external virtualization software.Although KVM, Xen, and QEMU based solutions are more successful against evasion methods relying on virtualization and emulation detection, they are not fully immune to hardware fingerprinting techniques used by malicious software. Besides, due to the platforms on which they are installed such as a specific operating system, they stand not a fully flexible solution especially for forensic and emergency response

	External Virtualization Software Dependency	Analysis on Bare Metal Machine	Turn Existing System to Analysis Environment	Machine Learning Based Classification Assistant
Cuckoo Sandbox	Virtualbox (in default)	No	No	No
CXPInspertor	KVM	No	No	No
Drakvuf	Xen	No	No	No
Anubis	QEMU	No	No	No
Ether	XEN	No	No	No
ISKRA	None	Yes	Yes	Yes

Table 7.1 Comparison of ISKRA and known and popular dynamic analysis platforms.

---

[1]See https://cuckoosandbox.org/

[2]See http://anubis.iseclab.org/

[3]See https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2012/11/26/TR-HGI-2012-002.pdf

applications[4]

Table 7.1 presents a qualitative comparison of ISKRA and other known and popular dynamic analysis platforms.

[4] The incident response operation may require the creation of a similar computing platform to that under attack. This, in turn, necessitates the installation of various software, which takes considerable time. Our framework requires only a single installation on the target system.

8. DISCUSSION

In this section, we will discuss the extent of the immunity of the ISKRA framework to evasion techniques. Considering the possible evasion methods against the ISKRA framework, one of the first points of attack is pertaining to the installation phase of the framework. Since the framework offers different usage areas, two different cases are possible during installation. The first case is to build a dynamic analysis environment in a freshly installed system. In this case, since there is assumed to be no existing malware in the system, there is no risk that prevents the correct installation of the framework.

The second case concerns running the framework on an infected system, for instance during an incident response operation. Malware running on a compromised device can disrupt the integrity of the system and change its normal behavior. Thus, it can compromise all interactions with the system. In such a case, it can monitor and prevent new software installations, including the ISKRA framework. However, this attack requires that malware perform attacks such as privilege escalation, which are complicated as we observe in targeted attacks. Also, such interventions to the system make the malware attack much more conspicuous. Thus, it contradicts with the primary aim of the type of malware studied here, which is to evade detection. For this reason, we think that common attacks that can be performed during the installation of ISKRA will be ineffective as they can be easily detected.

Once the correct installation is achieved, the ISKRA driver component will hide itself from malicious software by using the advantages of Intel VT-x, as explained previously. But, there is still a possibility that the agent component running in user mode will be targeted by attackers. For this, the ISKRA driver implements evasion techniques as it can monitor the outputs and return values of the system calls and alter them to remove data which indicates the presence of the framework. For instance, it deletes the process name of the ISKRA agent from the `NtQuerySystemInformation` system call output to remove it from the process list.

On the other hand, performing network and file operations, which are now done by

the ISKRA agent, in the driver component would be more costly in the VMX root mode than in the user mode. This is expected as the use of high-performance APIs offered by the operating system is quite limited. This would increase the running cost of the framework and cause the framework to consume more system resources. The increased usage of system resources may even allow malware to detect the framework by deploying side-channel analysis. Obviously, there is a trade-off here: make the ISKRA agent immune against attacks by running it in the VMX root mode, but accept heavy overhead as well as risk detection; and run it in the user mode and use evasion techniques to make it invisible to attackers, but still risk intervention by attackers. In this study, we opt for the latter.

9. CONCLUSION AND FURTHER WORK

Dynamic analysis platforms are one of the main tools used by cybersecurity experts to respond quickly to an increasing number of malware threats. However, evasion techniques used by current malware campaigns prevent these platforms from detecting malware and extracting the necessary indicator of compromise (IoC) information.

Based on the hypervisor-based monitoring system, the dynamic analysis platform proposed in this study, ISKRA, can record the behavior of malicious software without being detected as it runs at a higher level of privilege than malware. The existing solutions examined in Section 6.1 and Chapter 7 depend on large-scale platforms such as Xen, QEMU, or Virtualbox to create an analysis environment. But, ISKRA can work on all physical and virtualized systems and its only requirement is Intel VT-x support. Thanks to that it can be installed on an existing working system, this feature allows any computer with VT-x to be transformed into an automated analysis platform and supports analysis without being detected by the attacker during an incident response operation without the need for any changes in the codebase.

In this study, we experiment with very recent and the most evasive types of malware samples and show that none of them can detect the analysis framework. Therefore, they run, attempt to fulfill their mission and leaves all IoC behind, which is open to full inspection.

Also in our experiments, system call sequences collected by ISKRA are analyzed; and the results show that it is possible to detect malicious software without the need for any signature database with an AUC as high as 82%.

In the future, the analysis of collected system calls can be carried out in real-time as they are collected so that ISKRA will not only be able to bypass evasion techniques employed by malware and detect them, but also to prevent malware attacks in end-user machines. Besides, system call sequence groups can be examined and mapped to "MITRE ATT&CK" techniques. This will make it easier for analysts to understand and classify the behavior of malicious software.

BIBLIOGRAPHY

Chiappetta, M., Savas, E., & Yilmaz, C. (2016). Real time detection of cache-based side-channel attacks using hardware performance counters. *Appl. Soft Comput.*, *49*, 1162–1174.

Dinaburg, A., Royal, P., Sharif, M., & Lee, W. (2008). Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, (pp. 51–62)., Alexandria, Virginia, USA. Association for Computing Machinery.

Gadhiya, S. & Bhavsar, K. H. (2020). Techniques for malware analysis.

Ganesan, R., Murarka, Y., Sarkar, S., & Frey, K. (2013). Empirical study of performance benefits of hardware assisted virtualization. In *Proceedings of the 6th ACM India Computing Convention on - Compute '13*, (pp. 1–8)., Vellore, Tamil Nadu, India. ACM Press.

Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780.

Jadhav, A., Vidyarthi, D., & Hemavathy M. (2016). Evolution of evasive malwares: A survey. In *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, (pp. 641–646)., New Delhi, India. IEEE.

Korkin, I. & Tanda, S. (2017). Detect Kernel-Mode Rootkits via Real Time Logging & Controlling Memory Access. *arXiv:1705.06784 [cs]*. arXiv: 1705.06784.

Kulah, Y., Dincer, B., Yilmaz, C., & Savas, E. (2019). SpyDetector: An approach for detecting side-channel attacks at runtime. *International Journal of Information Security*, *18*(4), 393–422.

Lengyel, T. K., Maresca, S., Payne, B. D., Webster, G. D., Vogl, S., & Kiayias, A. (2014). Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, (pp. 386–395)., New Orleans, Louisiana, USA. Association for Computing Machinery.

Lundsgård, G. & Nedström, V. (2016). Bypassing modern sandbox technologies.

Manky, D. (2013). Cybercrime as a service: a very modern business. *Computer Fraud Security*, *2013*(6), 9 – 13.

Milošević, N. (2014). History of malware. *arXiv:1302.5392 [cs]*. arXiv: 1302.5392.

Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, (pp. 421–430). ISSN: 1063-9527.

Schultz, M., Eskin, E., Zadok, F., & Stolfo, S. (2001). Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S P 2001*, (pp. 38–49). ISSN: 1081-6011.

Sikorski, M. & Honig, A. (2012). *Practical malware analysis: the hands-on guide to dissecting malicious software*. San Francisco: No Starch Press.

Stakhanova, N., Couture, M., & Ghorbani, A. A. (2011). Exploring network-based malware classification. In *2011 6th International Conference on Malicious and Unwanted Software*, (pp. 14–20).

Uppal, D., Mehra, V., & Verma, V. (2014). Basic survey on Malware Analysis,

Tools and Techniques. *International Journal on Computational Science & Applications*, *4*(1), 103–112.

Vinayakumar, R., Soman, K., Poornachandran, P., & Sachin Kumar, S. (2018). Detecting Android malware using Long Short-term Memory (LSTM). *Journal of Intelligent & Fuzzy Systems*, *34*(3), 1277–1288.

Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., & Rossow, C. (2016). SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In Monrose, F., Dacier, M., Blanc, G., & Garcia-Alfaro, J. (Eds.), *Research in Attacks, Intrusions, and Defenses*, Lecture Notes in Computer Science, (pp. 165–187)., Cham. Springer International Publishing.