A Systematical and Longitudinal Study of Evasive Behaviors in Windows Malware

Nicola Galloro[a], Mario Polino[a], Michele Carminati[a], Andrea Continella[b], Stefano Zanero[a]

[a]*DEIB, Politecnico di Milano,*
[b]*EEMCS, University of Twente,*

Abstract

Malware is one of the prevalent security threats. Sandboxes and, more generally, instrumented environments play a crucial role in dynamically analyzing malware samples, providing key threat intelligence results and critical information to update detection mechanisms.

In this paper, we study the evasive behaviors employed by malware authors to hide the malicious activity of samples and hinder security analysis. First, we collect and systematize 92 evasive techniques leveraged by Windows malware to detect and thwart instrumented environments (e.g., debuggers and virtual machines). Then, we implement a framework for evasion analysis of x86 binaries and analyze 45,375 malware samples observed in the wild between 2010 and 2019; we compare this analysis against popular, legitimate Windows programs to study the intrinsic characteristics of such evasive behaviors.

Based on the results of our experiments, we present statistics about the adoption of evasive techniques and their evolution over time. We show that over the past 10 years, the prevalence of evasive malware samples had a slight increase (12%). Moreover, the employed techniques shifted significantly over time. We also identify techniques that are specific to malware, as opposed to being employed by both malicious and legitimate software. Finally, we study how the security community reacts to the deployment of new evasive techniques. Overall, our results empirically address open research questions and provide insights and directions for future research.

1. Introduction

Malware is one of the most dangerous and prevalent Internet threats. Financially-motivated threats, such as ransomware [1, 2], banking trojans [3], and cryptominers [4, 5, 6], have dominated the malicious landscape in the past years [7, 8]. Given the complexity of applying static analysis to modern malware samples, especially when heavily obfuscated [9], malware analysts often rely on dynamic analysis to reverse-engineer new samples, providing key threat intelligence results and critical information to update detection mechanisms. In fact, dynamic analysis is starting to be also employed in malware detectors, some of which leverage emulators to execute suspicious binaries in an instrumented environment [10].

Previous literature studied how malicious samples employ techniques to recognize when they are executed in a controlled environment and subsequently hide or alter their malicious behavior. The detection of the analysis environments is performed through *fingerprinting*: Malware samples look for specific artifacts left by the analysis components (or agents). If any such artifact is identified, the malicious activity is not executed (or executed differently) [11, 12, 13, 14, 15]. Different solutions have been proposed to address this problem. Multiple path exploration approaches force the execution of the whole program to discover hidden behaviors but suffer from the well-known path explosion problem [16]. An alternative is to detect inconsistencies between the execution of a sample in a sandbox and the execution of the same sample without the instrumentation layer. This approach has been tested by Kirat et al. [17], who measured the difference between bare-metal and sandbox executions to detect "paths" related to evasion techniques. However, this method does not intercept low-level system interactions, such as system calls and x86 instruction, because the bare-metal setup can only reason about raw disk content. A more recent solution is to execute the malicious samples inside a Dynamic Binary Instrumentation (DBI) framework that hides the presence of the analysis environment, identifying anti-instrumentation techniques [18, 19, 20].

This paper provides the first comprehensive, systematic, and longitudinal study of the evasive behaviors adopted by Windows malware. Our research attempts to answer important questions that are currently still open: What are the prevalent evasion techniques adopted by malware samples to hinder security analysis? How have such techniques evolved in the past years? How does the security community react to the adoption of a new evasive

technique? Are evasive techniques also adopted by legitimate programs? To answer these questions, we first collect and systematize 92 evasive techniques adopted by Windows malware. To the best of our knowledge, we provide the most extensive taxonomy of evasive techniques: The previous most complete surveys [21, 22] cover 41 techniques. Then, we develop a fine-grained framework to identify and collect information about the evasive behaviors present in Windows programs. Built on top of Arancino [18], our framework leverages the capabilities of a DBI tool, Intel Pin, to instrument the target executable and identify when the program under analysis attempts to collect artifacts to evade the analysis itself.

Using our framework and taxonomy, we study the evasive behaviors adopted by 45,375 malware samples observed in the wild between 2010 and 2019. We identify the most common evasive techniques and correlate them with the different malware families, providing insights about different malware campaigns. To further investigate the characteristics of such evasive techniques, we test their adoption in popular, legitimate Windows programs. Our longitudinal analysis indicates that the amount of evasive malware samples has not increased significantly in the past 10 years. However, the techniques adopted by these evasive samples varied over time.

In summary, we make the following contributions:

- We systematically study evasive behaviors employed by Windows malware by collecting 92 techniques used to evade instrumentation environments (e.g., Debuggers, Virtual Machines)—the largest collection so far—and providing a taxonomy to classify such techniques according to their semantics and characteristics.

- We analyze 45,375 malware samples observed in the wild and 949 popular, legitimate applications. We study the correlation between malware families and the adopted evasive techniques, and we analyze both malicious and legitimate programs to compare their evasive behaviors.

- We perform a longitudinal study of the adoption of evasive techniques in malware families between 2010 and 2019, and we provide statistics and insights about the evolution of evasive behaviors. Additionally, we empirically estimate how the community reacted to and influenced the adoption of evasive techniques during the same period. For instance, we show that the adopted techniques varied in the past years, while the number of evasive samples is rather stable.

Table 1: Comparison of large-scale studies of evasive malware techniques: ● dynamic analysis, ○ static analysis, ✓ or ✗ if the work provides or not, respectively, the considered analysis.

	Year	*Taxonomy*	Dynamic-Static Analysis	Longit. Analysis	Malware vs Goodware	# Evasive Techniques	# Samples (# Families)
Chen et al. [26]	2006	✓	●	✗	✗	8	6,222 (✗)
Lindorfer et al. [14]	2011	✓	●	✗	✗	14	1,500 (175)
Branco et al. [27]	2012	✗	○	✗	✗	51	4M (✗)
Barbosa et al. [28]	2014	✗	○	✗	✗	51	12M(✗)
Polino et al. [18]	2017	✗	●	✗	✗	24	7006 (✗)
Miramirkhani et al. [15]	2017	✓	●	✗	✗	44	✗
Oyama [29]	2018	✗	●	✗	✗	28	8243 (3044)
Afianian et al. [21]	2019	✓	✗	✗	✗	46	✗
D'Elia et al. [19]	2020	✓	●	✗	✗	45	1000 (✗)
Chen et al. [30]	2021	✓	○	2009-2014	✗	41	17,283 (6)
This work	2021	✓	●	2010-2019	✓	92	45,375(2867)

2. Related Work

The analysis of the evasive techniques by which malware determines the presence or absence of an analysis system is a wide research topic for which we refer the reader to [21, 22] that provide comprehensive surveys and taxonomies of 41 existing anti-analysis techniques. Unlike our work, which quantifies the adoption of 92 evasive techniques from the dynamic execution of actual malware, they put in place a qualitative analysis only. In this section, we describe the related works that investigate the evasive behavior in a large-scale study of malware found in the wild. Table 1 provides a comparison of the recent works in the area, according to the type of analysis they conducted. Regarding the analysis of evasive techniques used by legitimate non-malicious software vendors in order to prevent analysis or piracy of their software, we refer the reader to [23, 24, 25].

Chen et al. [26] proposed a first taxonomy of malware anti-analysis techniques by comparing execution traces from different environments. They found that 2.7% of 6,222 samples from 2006-2007 exhibited less behavior when running in a VM versus a bare-metal system.

Branco et al. [27] and Barbosa et al. [28] introduced various static detection methods for anti-debugging and anti-VM techniques and run analysis over 4 and 12 million samples, respectively, to show the state of evasion techniques in use. In their study, anti-analysis operations were detected only through static analysis, while dynamic behavior was not considered. The main problem with static analysis resides in packing and obfuscation

techniques, which are widely used by malware authors precisely to avoid it. Indeed, since static analysis is based on parsing and comparing executable byte sequences, obfuscated samples are opaque to it. Additionally, there is no guarantee that any matched technique is actually executed. On the contrary, with dynamic analysis, we can precisely identify evasive techniques while they are executed, even when packing and obfuscation are applied. In summary, dynamic analysis is more precise in this context. Compared to static analysis, our approach—and dynamic analysis in general—is resilient to packing due to the dynamic nature of the DBI analysis that allows a profound control over the instrumented binary.

Chen et al. [30] surveyed anti-VM and anti-debugging techniques in 17,283 "generic" and targeted (APT) malware samples collected between 2009-2014. The authors found that these techniques became more prevalent over the surveyed timeframe and that their presence was negatively correlated with antivirus detection. They also observed that APTs generally do not use as many anti-analysis techniques as generic malware and that they have decreased their use over time. Although their study is similar to the present one, since they provide a longitudinal study of the trends in anti-analysis techniques analyzing malware samples, there are also many differences. First, in their study, anti-analysis techniques were detected only through static analysis, while dynamic behavior was not considered. Therefore, it shares the same limitations of [27, 28] presented before. Moreover, they limit their analysis to only six malware families without clarifying in detail the techniques executed by each of them. Consequently, their results are biased by the restricted number of families considered. In the present work, instead, we strive to select malware samples equally distributed in time.

Polino et al. [18] found that 15.6% of 7,006 malware samples collected from VirusTotal between October 2016 and February 2017 exhibited anti-instrumentation behavior (not necessarily solely anti-DBI). Similarly to the present work, the authors leverage the capabilities of a DBI tool, Intel Pin, to instrument the target executable and identify when the program under analysis attempts to collect artifacts to evade the analysis itself. However, they do not consider Anti-VM and Anti-Debugging behaviors.

Miramirkhani et al. [15] assesses the threat of malware sandbox evasion strategies that leverages artifacts indicative of the "wear and tear" and "aging" of a system to identify artificial environments and demonstrate its effectiveness against existing sandboxes used by malware analysis services. Commonly to this work, as presented in Tables 3 and 4, we check the "wear-and-

tear" state of the machine. However, while we focus on studying the evasion techniques currently exploited by malware, this work, instead, proposes novel evasive techniques against the existing sandbox mechanism.

Oyamama [29] analyzes a dataset of dynamic malware analysis results and examines trends in the Anti-Analysis operations executed by malware samples collected in 2016. Among 8243 malware samples, 856 (10.4%) samples executed at least one type of the 28 Anti-Analysis operations investigated.

D'Elia et al. [19] present a DBI-based prototype that, similarly to Arancino [18] and our work, offers a stealthy execution environment based on an observe-check-replace design to intercept evasive attempts and hide imperfections. They demonstrate its effectiveness against a set of 45 highly evasive samples.

In contrast to the presented studies, in this work, we perform a longitudinal study of the adoption of evasive techniques by dynamically executing malware to provide quantitative results and insights into their evolution and adoption. Moreover, we explore the adoption of evasive behaviors in legitimate programs to investigate whether evasive techniques are intrinsically evasive or also used for legitimate purposes. Finally, we analyze a higher number of samples and techniques.

3. Taxonomy of Evasive Techniques

An evasive technique is a specific activity performed by a malware sample to detect a dynamic analysis system (and subsequently thwart it). In particular, we focus on any activity, under the form of system calls, Windows API functions, memory accesses, or x86 instructions, that leaks the presence (i.e., artifacts) of analysis environments. We study evasive techniques targeting virtual machines and sandboxes (i.e., Anti-VM techniques), as well as debuggers and DBI implementations (i.e., Anti-Debugging techniques). Note that some techniques can overlap this broad categorization because of common artifacts leaked by both classes of environments.

After gathering and analyzing 92 evasive techniques found in the literature, we propose to classify them according to the semantics of the action(s) performed on the OS. Proceeding thus, starting from systematization previously presented [15, 20, 26, 14, 21, 30], we develop a taxonomy consisting of 16 semantic-equivalent groups. In the following, we list the groups, providing a few examples of techniques that belong to each. For further details on each technique, we refer the reader to Appendix A.

Memory Fingerprinting. These techniques access memory regions of the running process to detect debuggers. In fact, debuggers modify several system variables associated with their target process. For instance, the `PEB`, a structure present in each Windows process, contains variables designed to indicate the presence of a debugger.

Exception Handling. With this type of approach, a malware sample throws exceptions that a debugger handles for debugging purposes. The malicious sample sets up a handling routine for the exception and, if the routine is never executed, it means that the debugger caught the exception. A less trivial example is `NtClose(INVALID_HANDLE)`: Closing an invalid handle causes an exception only if there is a debugger attached to the process.

CPU Fingerprinting. This category contains all the Anti-VM techniques that fingerprint the CPU, detecting differences between physical CPUs and virtualized ones. This behavior can be intended, or it can be a side-effect due to incorrect execution of sensitive instructions by the hypervisor. An example is `CPUID`, an instruction that retrieves information and features from the CPU and returns the presence of a hypervisor and its brand if present [31].

Table Descriptors. This category contains x86 instructions used to retrieve the addresses of OS Table Descriptors. These addresses frequently change if a hypervisor is present. Thus they have been used to detect virtualized environments. Nevertheless, these techniques are viable only on single-core machines—they are not useful with modern systems.

Traps. Another way to throws exception is the use of x86 trap instructions. As a matter of fact, `INT 3` is the typical breakpoint instruction intercepted by a debugger. If this instruction is present in a non-debugged program, the program receives the corresponding signal. A similar concept applies to VM trap instructions like `VPCEXT`.

Timing. These techniques contain all the methods to precisely measure time-flow and CPU clock ticks. These measurements are used by malware to spot the presence of slowdowns during the execution of malicious code. Most analysis systems have a significant impact on performance. These slowdowns are easily detectable by a malware sample. For example, it is possible to measure the number of ticks elapsed during the execution of a set of instructions using the `RDTSC` instruction. If this instruction triggers a virtual machine exit, the number of ticks is significantly higher.

Stalling. This category contains all the techniques that put the executable in sleep mode to avoid detection. These techniques work because sandboxes

have limited time allocated for the analysis, so they could miss malicious activities of malware samples that activate after a long period. Sleeping behaviors can fool even debuggers; a loop that continuously invokes sleeping routines is frustrating to step through during manual debugging.

Human Interaction. In an automated system where thousands of malware samples are analyzed inside virtual machines, there is no human interaction. There is a category of evasion techniques that exploit this absence of interaction to identify the instrumented environment. For example, a malware sample can assume that the environment is instrumented if the mouse cursor is not moving.

Registry. Some malware samples access the Windows Registry searching for the virtual machine or debuggers-related artifacts such as *VirtualBox Guest Additions*. This class of techniques considers the use of a blacklist of resources that should not be accessed.

System Environment. This class contains all the techniques that detect system-related information that can reveal the presence of a VM or debugger. For instance, an using `GetAdaptersInfo` to retrieve the MAC information. MAC address or graphic devices have a manufacturer-fixed part that is well known for virtual machines.

WMI. An extension of the System Environment category is the WMI class. WMI is a Microsoft technology for Windows machines maintenance. Using the WMI framework and the corresponding query language `WQL`, it is possible to query any information about Windows machines' configuration and change their settings. Evasive malware can exploit these capabilities to detect analysis tools or installed software.

Process Environment. This category refers to techniques used to collect information about the running process. An example is `NtGetContextThread(CONTEXT_DEBUG_REGISTERS)`: this API call retrieves the values of hardware debug registers, employed to perform hardware debugging.

File System. This category includes all those techniques that rely on artifacts present on the file system (e.g., the presence of a Python-based agent, `agent.py`, in the Home folder).

List Processes. Malware can enumerate the running processes to identify an agent process (e.g., `python agent.py`).

List Services. These techniques enumerate the active services to identify an analysis service.

Drivers. Malicious samples can enumerate the list of drivers to identify

emulated or virtualized devices.

Complexity of the Evasive Techniques. We study the complexity of the techniques mentioned above and the difficulty of their implementation. To do so, for each of the 92 techniques, we implement a program that leverages a given technique to detect when it is running in a controlled environment, i.e., returning True or False. By statically analyzing each produced binary program, we count the number of basic blocks, instructions, and function calls required to implementation the 92 evasive techniques (columns **BB**, **I**, and **C** in Tables 3 and 4). We also report whether the implementation of each technique, at the source-code level, requires writing assembly code, which indicates more advanced skills (column **ASM** in Tables 3 and 4). While none of the studied techniques require specific privileges or can be considered complex or challenging to implement, some groups of techniques certainly result in smaller portions of code. For instance, Memory Fingerprinting techniques, although they require writing assembly code, are implemented with a few instructions (8 to 16) and without invoking any additional functions. On the contrary, List Processes and List Services techniques result in larger portions of code (up to 340 instructions) containing several function calls (up to 27).

4. Analysis Framework

To perform our study, we developed an automated analysis framework to analyze the evasive behaviors of Windows programs. Following a recent research trend [18, 20, 19], which demonstrated the effectiveness of DBI tools in defeating evasion attempts, we leverage Intel Pin to fully control the execution flow of the instrumented program. In fact, by instrumenting a binary, we can identify when it attempts to spot artifacts to evade the analysis. Our framework extends a previously proposed system [18], and it can instrument executables at four levels: (1) Instruction (monitor each executed instruction); (2) API Hook (hook interesting Windows APIs); (3) Syscall (hook interesting syscalls); (4) Memory Access (monitor access to specific memory regions).

Leveraging these four instrumentation levels, we implemented a set of countermeasures that dismantle all the 92 presented evasion techniques. Our system inserts two hooks before and after instructions or function calls that can be associated with an evasive technique. The first hook is executed *before* the evasion while the second *after* it. Both hooks contain a logging routine

that logs the evasive behavior and applies hiding countermeasures. We are able to insert hooks with such controls at the instruction level, leveraging Just-In-Time recompilation of Intel's PinTool. Our system has four main modules: the **System Calls Hooker** and the **Windows API Hooker**, which detect and apply the countermeasures of evasive techniques based on system calls and Windows APIs, and the **Memory Accesses Controller** and the **x86 Instructions Monitor** for evasion based on memory reads and x86 instructions. Moreover, we implemented a **Library Tracer** module to log the call stack of the evasive technique and related libraries. Windows libraries execute some instructions, API calls, or syscalls for legitimate purposes. Therefore, if we consider only their presence, we cannot claim that the malware is evasive. Using the Library tracer, we can detect which activities are executed from the malware text segment.

It is important to note that our system does not stop its analysis after the first detected technique. In the case of malware samples that implement a sequence of multiple evasions attempts, our system detects the first technique executed and applies the corresponding bypass mechanism described before, avoiding that the malware triggers evasive behavior. In this way, our system can analyze and identify any secondary techniques executed after the first "barrier."

A common bypass mechanism employed throughout our analysis consists of hiding artifacts that are commonly present in sandboxes. As an example, whenever a malware samples accesses file information executing syscalls like `NtOpenFile`, `NtCreateFile`, `NtOpenKey`, or `NtQueryAttributesFile`, our framework returns a `NOT_FOUND` response if the file path contains suspicious keywords defined in Table 5 (e.g., "virtualbox"). In other words, our system logs the attempt to retrieve information about the environment and ensures that the API call returned value corresponds to a not existing key. A similar countermeasure is applied when the malware accesses *Keys* inside the Windows Registry or enumerates drivers, Windows services, and processes. Differently, a relevant bypass mechanism involves reducing the value returned by time measurement instructions such as `RDTSC` to eliminate the overhead introduced by analysis environments. We include an extensive discussion of our evasion countermeasures in Appendix A. Moreover, our countermeasures follow the directions established in previous work [18, 20, 19].

Validation. We manually validated our framework by developing "test" programs and verifying that our framework could identify the usage of all

the 92 (known) techniques described in Section 3. Moreover, we tested our detection system against Pafish [32], a known tool that implements several evasive techniques. Our framework was able to detect all the techniques implemented by Pafish, remaining undetected. Naturally, this does not guarantee that our framework can detect all the malware evasive behavior that malware can adopt, as it is an undecidable problem. We discuss our limitations in Section 6.

5. Evasive Behavior Analysis

In this section, we describe our experimental setup and the results of our study, which specifically aims to answer the following research questions.
RQ1. *What are the most common evasive techniques adopted by malware?* We analyze 45,375 samples from 2,867 different families and report insights about the adopted evasive technique together with the specific keywords searched by samples to spot analysis artifacts, e.g., "vbox" (Section 5.2).
RQ2. *Is there a correlation between malware families and the adopted evasive techniques?* We study this correlation by training and evaluating a set of classifiers that aim at distinguishing malware families by looking at the set of adopted techniques (Section 5.3).
RQ3. *Has the number of adopted evasive techniques, per family and per category, changed over the past 10 years? When did each evasive technique appear for the first and last time?* We perform a longitudinal study over the past 10 years, and we provide insights into the evolution of evasive behaviors over time (Section 5.4).
RQ4. *How does the adoption of malware evasion techniques impact on the security community, and vice versa?* We harvest papers, reports, and blog posts from the security community estimating the influence of the public disclosure of evasive techniques on their adoption in the wild (Section 5.5).
RQ5. *Are evasive techniques also adopted by legitimate software?* We analyze 516 legitimate programs and study evasive behaviors in goodware. We further compare such results with the data obtained from the analysis of malware samples (Section 5.6).

5.1. Datasets & Setup

We leverage two datasets: a malware dataset and a goodware dataset. For the former, we collected 45,375 malware samples from VirusTotal, distributed in a timeframe of 10 years. Samples are ordered by the first appearance

date on VirusTotal. In particular, we downloaded and analyzed more than 1,000 samples per quarter, from January 2010 to September 2019, in order to perform a longitudinal, time-based analysis of the evasion phenomenon. We collected only samples that were classified as malicious by at least 35 different AV scanners. Such a conservative choice is coherent with what a recent study suggested to mitigate the known label dynamics issue [33]. A lower number of AV detections would have caused the download of possible false positives, which would have biased our analysis. Nonetheless, we acknowledge that, as a consequence of this choice, our dataset might not include samples that are not well detected by the majority of endpoint solutions (Section 6).

We leveraged `AVClass` [34] to obtain the malware family of each sample. Our dataset contains samples belonging to 2,867 unique families, while 1,369 samples (3%) were not tagged in any malware family by `AVClass`—we included these samples in our experiments, but we did not consider them when studying per-family trends. Figure 1 shows the number of families detected in each quarter. The top 1,000 families account for almost the whole (93%) dataset, while the top 100 families account for around 67% of the dataset.

For the benign dataset, we downloaded 327 executables from PortableApps.com and 622 generic PE files from a variety of Windows OS executables. In total, we collected 949 benign executables. All the benign applications were executable desktop applications. None of our benign samples was installed from Windows Store. We filtered out any executables that were detected positives by at least one anti-malware on VirusTotal. After this process, our dataset was composed of 516 goodware binaries. All our executables were compatible with Windows 7 and later versions.

Experimental Setup. We deployed our analysis framework on 6 Windows 10 VirtualBox VMs. Each VM is managed by our analysis manager, which spawns the VM, sends the sample to the VM agent in charge of the DBI analysis, and collects the evasion report. After each execution, we rolled back the virtual machine to a clean snapshot. As done in the previous work [18], we let each sample run for up to 5 minutes, a reasonable time to trigger most of the evasive behaviors that our tool can detect. In fact, a recent study [35] showed that most of the behaviors manifested by malicious samples in a sandbox (and 98% of the executed basic blocks) are observed during the first two minutes of execution. Finally, following the best practices for malware experiments [36], we allowed the samples to communicate with their control servers and denied any potentially harmful traffic (e.g., spam) during the

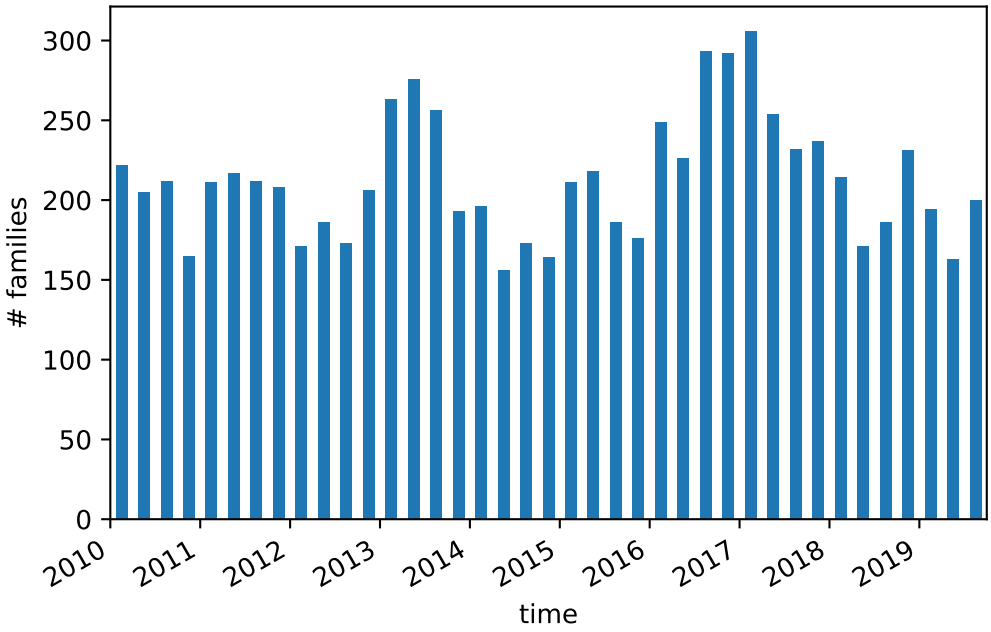


Figure 1: Number of families over time

experiments. Our analysis machine runs on a separate local network with no other machine connected, thus, self-spreading malware does not have any local target. To support scientific repeatability, we will release our analysis framework.

5.2. Adoption of Evasive Techniques

Among the 92 collected techniques, we measured the percentage of techniques adopted during each considered year. Tables 3-4 show statistics about the presence of each technique in our malware datasets, considering also the taxonomy introduced in Section 3. Our data show a significant usage of Timing and Stalling techniques in both malware and goodware, which use them for synchronization and timing purposes. Also, an interesting result is that 15 techniques described in the literature have never been observed in our dataset, neither by malware nor goodware samples. These techniques can be used for evasion purposes, but they do not seem actively exploited by our collected samples.

Table 2 contains data about the evasiveness of the top 20 most prevalent families. Collected metrics show that most of them first appeared in 2010 and that not always the most prominent family is the most evasive.

Table 2: Top 20 families. # refers to the number of samples, **Date** refers to the first appearance in VirusTotal of a sample belonging to the specified family, **# Ev.** is the number of evasive samples exposing at least a technique, **# Packed** is the number of samples that are packed according to PEiD, **# Tech.** is the maximum number of techniques detected in a sample of the related family.

Family	**#**	**Date**	**# Ev.**	**# Packed**	**# Tech.**
zbot	2412	01/2010	1564 (65%)	601 (25%)	19
vtflooder	1016	06/2014	991 (98%)	1 (0%)	15
installerex	859	01/2012	850 **(99%)**	2 (0%)	14
firseria	835	08/2011	801 (96%)	91 (11%)	11
multiplug	807	03/2012	782 (97%)	93 (12%)	14
fareit	750	08/2011	624 (83%)	506 (67%)	**21**
domaiq	743	02/2013	741 **(99%)**	67 (9%)	7
high	652	01/2010	652 (53%)	552 (45%)	19
vobfus	570	01/2010	227 (40%)	444 (78%)	15
delf	558	01/2010	490 (88%)	462 (83%)	17
flystudio	550	01/2010	474 (86%)	474 (86%)	20
dealply	485	12/2010	424 (87%)	433 (89%)	15
neshta	482	06/2011	470 (98%)	450 **(93%)**	17
installcore	474	06/2014	356 (75%)	330 (70%)	10
gamarue	473	10/2011	348 (74%)	273 (57%)	11
xorist	447	10/2010	48 (11%)	15 (3%)	12
loadmoney	440	12/2012	367 (83%)	62 (14%)	14
autoit	436	01/2010	420 (96%)	28 (6%)	18
bladabindi	404	02/2011	83 (21%)	343 (85%)	15
virlock	402	11/2014	272 (68%)	1 (0%)	15

In terms of evasion, the most evasive families are `installerex` and `domaiq`, with a striking 99% of evasive samples, immediately followed by `vtflooder` and `neshta`, with 98% of evasive samples. This aspect often pairs with the maximum number of different techniques detected in a sample of the related family. Indeed, the `installerex`, the `vtflooder`, and the `neshta` families contain samples employing at least 15 different evasive techniques. The only exception is represented by `domaiq` that, instead, employs up to 7 different techniques. Interestingly, `neshta` is also the family with the highest number of samples that are packed according to PEiD. Table 2 also shows rarely evasive families such as `xorist` and `bladabind`, with 11% and 21% of evasive samples, respectively. Another interesting takeaway is that families like `fareit` and `flystudio` have few samples exposing many evasive techniques. Figure 2 shows the normalized quarterly distribution of samples employing evasive techniques for each category. On average, the most common exploited categories are Timing and Stalling. Search for specific drivers is very rare,

Table 3: Techniques **D/V**: ◐Anti-Debugging, ◑Anti-VM, ●Both. ◇Used by Goodware. **BB** is the number of basic blocks, and **I** is the number of instructions, **C** is the number function calls, **ASM** indicates whether implementing the given technique requires writing assembly code.

		Technique	#Families (%)	#Goodware (%)	#Malware (%)	D/V	G.	First	Last	BB	I	C	ASM
Memory Fingerprinting		PEB→IsDebugged	162 (5.65)	72.0 (7.58)	982 (2.16)	◐	◇	01/10	09/19	1	9	0	Y
	[37]	KUSER_SHARED→KdDebuggerEnabled	23 (0.8)	0 (0)	102 (0.22)	◐		01/11	08/19	1	8	0	Y
	[26]	PEB→NtGlobalFlag	46 (1.6)	0 (0)	96 (0.21)	◐		02/10	03/19	1	9	0	Y
	[38]	PEB→Heap→Flags	2 (0.07)	0 (0)	3 (0.01)	◐		05/17	02/19	4	16	0	Y
		PEB→Heap→ForceFlags	2 (0.11)	0 (0)	3 (0.01)	◐		09/16	05/17	1	10	0	Y
Exception Handling		SetUnhandledExceptionFilter	1817 (63.38)	475.0 (50.0)	15651 (34.49)	◐	◇	01/10	09/19	8	27	3	Y
		NtClose(INVALID_HANDLE)	606 (21.14)	19 (2.0)	3105 (6.84)	◐	◇	01/10	09/19	14	47	6	N
	[39]	OutputDebugString	234 (8.16)	26.0 (2.74)	794 (1.75)	◐	◇	01/10	08/19	14	55	6	N
		POPF/D - TRAP FLAG	58 (2.02)	0 (0)	180 (0.4)	◐		01/10	06/19	9	38	4	Y
		CTRL Exception Handling	0 (0)	0 (0)	0 (0)	◐		Never	Never	10	34	4	N
CPU Fingerprinting		CPUID(EAX=0x00000001)	1333 (46.49)	320.0 (33.68)	10070 (22.19)	◑	◇	01/10	09/19	1	11	0	Y
	[40]	IN	234 (8.16)	0 (0)	869 (1.92)	◑		01/10	09/19	12	51	5	Y
	[41]	CPUID(EAX=0x40000000)	31 (1.08)	0 (0)	119 (0.26)	◑		12/10	08/19	23	171	7	Y
		STR	50 (1.74)	0 (0)	122 (0.27)	◑		01/10	09/18	5	18	2	Y
		SMSW	9 (0.31)	0 (0)	11 (0.02)	◑		02/10	06/15	5	18	2	Y
Table Descr.		SLDT	37 (1.29)	0 (0)	176 (0.39)	◑		01/10	08/19	1	5	0	Y
	[41]	SGDT	16 (0.56)	0 (0)	126 (0.28)	◑		03/11	08/19	6	28	1	Y
		SIDT	46 (1.6)	0 (0)	162 (0.36)	◑		01/10	04/17	6	22	1	Y
Traps		INT 3	172 (6.0)	0 (0)	747 (1.65)	◐		01/10	08/19	11	38	4	Y
	[39]	VPCEXT	105 (3.66)	0 (0)	362 (0.8)	◑		01/10	09/19	14	52	6	Y
	[42]	POP SS	92 (3.21)	0 (0)	289 (0.64)	◐		01/10	05/19	1	14	0	Y
	[41]	INT 1	74 (2.58)	0 (0)	141 (0.31)	◐		02/10	08/19	11	37	4	Y
	[37]	ICEBP	27 (0.94)	0 (0)	61 (0.13)	◐		01/10	09/19	11	37	4	Y
	[43]	INT 2D	10 (0.35)	0 (0)	39 (0.09)	◐		03/10	05/19	11	38	4	Y
Timing		GetTickCount	1464 (51.06)	198.0 (20.84)	11029 (24.31)	●	◇	01/10	09/19	9	34	3	N
		RDTSC/D	1398 (48.76)	168.0 (17.7)	9518 (20.98)	●	◇	01/10	09/19	31	231	3	Y
		QueryPerformanceCounter	994 (34.67)	140.0 (14.74)	6038 (13.31)	●	◇	01/10	09/19	41	243	6	N
		GetLocalTime	587 (20.47)	75.0 (7.89)	2707 (5.97)	●	◇	01/10	09/19	9	41	3	N
		timeGetTime	217 (7.57)	32 (3.37)	805 (1.77)	●	◇	01/10	09/19	3	10	1	N
		NtQuerySystemTime	174 (6.07)	20.0 (2.11)	667 (1.47)	●	◇	01/10	09/19	9	50	3	N
		GetSystemTimes	22 (0.77)	0 (0)	37 (0.08)	●		07/13	09/19	9	43	3	N
		KUSER_SHARED→SystemTime	13 (0.45)	0 (0)	27 (0.06)	●		01/13	09/19	6	61	1	Y
	[18]	KUSER_SHARED→InterruptTime	11 (0.38)	0 (0)	19 (0.04)	●		04/10	06/19	6	61	1	Y
	[41]	timeGetSystemTime	3 (0.1)	0 (0)	6 (0.01)	●		07/10	12/17	9	43	3	N
	[44]	KUSER_SHARED→TickCountQuad	12 (0.42)	0 (0)	17 (0.04)	●		09/10	05/17	6	30	1	Y
		NtGetTickCount	0 (0)	0 (0)	0 (0)	●		Never	Never	13	44	5	N
		QueryInterruptTime	0 (0)	0 (0)	0 (0)	●		Never	Never	15	67	5	N
		NtQueryPerformanceCounter	0 (0)	0 (0)	0 (0)	●		Never	Never	36	230	7	N
		QueryUnbiasedInterruptTimePrecise	0 (0)	0 (0)	0 (0)	●		Never	Never	15	67	5	N
		QueryInterruptTimePrecise	0 (0)	0 (0)	0 (0)	●		Never	Never	15	67	5	N
		GetTickCount64	0 (0)	0 (0)	0 (0)	●		Never	Never	15	61	5	N
		QueryUnbiasedInterruptTime	0 (0)	0 (0)	0 (0)	●		Never	Never	15	67	5	N
Stalling		waitForSingleObject/Ex	1291 (45.03)	242.0 (25.47)	9117 (20.09)	●	◇	01/10	09/19	9	36	4	N
		Sleep/SleepEx	1249 (43.56)	41.0 (4.32)	9135 (20.13)	●	◇	01/10	09/19	3	10	1	N
		SetTimer	842 (29.37)	100.0 (10.53)	5686 (12.53)	●	◇	01/10	09/19	14	45	4	N
	[45]	SetWaitableTimer/Ex	75 (2.62)	1.0 (0.11)	195 (0.43)	●		01/10	09/19	7	46	3	N
	[46]	CreateTimerQueueTimer	23 (0.8)	2.0 (0.21)	92 (0.2)	●		04/11	02/19	20	64	6	N
		NtDelayExecution	22 (0.77)	37.0 (3.89)	130 (0.29)	●	◇	07/10	05/19	7	39	3	N
		timeSetEvent	33 (1.15)	6.0 (0.63)	84 (0.19)	●		02/10	08/19	5	22	1	N
		icmpSendEcho/2/Ex	6 (0.21)	1.0 (0.11)	42 (0.09)	●		11/13	07/19	16	54	4	N
HI	[47]	GetCursorPos	754 (26.3)	99.0 (10.42)	4157 (9.16)	◑	◇	01/10	09/19	10	32	3	N
	[48]	GetLastInputInfo	49 (1.71)	1.0 (0.11)	305 (0.67)	◑		04/10	09/19	6	26	3	N
Registry		NtOpenKey/Ex	193 (10.55)	12.0 (1.26)	781 (3.13)	●	◇	01/13	09/19	24	178	5	N
	[49]	NtEnumerateKey	215 (7.5)	2.0 (0.21)	790 (1.74)	●		01/10	09/19	69	334	18	N
		NtQueryValueKey	93 (3.24)	9.0 (0.95)	304 (0.67)	●	◇	03/10	09/19	16	70	5	N
		NtEnumerateValueKey	6 (0.21)	0 (0)	58 (0.13)	●		07/13	10/18	19	78	3	N
System Environment		NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)	1342 (46.81)	157.0 (16.53)	10651 (23.47)	◑	◇	01/10	09/19	15	53	4	N
		FindWindow	520 (18.14)	51.0 (5.37)	2245 (4.95)	●	◇	01/10	09/19	10	34	3	N
		GetComputerName	403 (14.06)	18.0 (1.9)	2041 (4.5)	◑	◇	01/10	09/19	15	50	2	N
		IsDebuggerPresent	304 (10.6)	17.0 (1.79)	1516 (3.34)	◐		01/10	09/19	6	14	1	N
	[50]	GetAdaptersInfo	165 (5.76)	2.0 (0.21)	1328 (2.93)	◑		01/10	09/19	54	136	13	N
	[40]	GetDiskFreeSpace/Ex	184 (6.42)	47.0 (4.95)	528 (1.16)	◑	◇	01/10	08/19	7	30	1	N
	[41]	CheckRemoteDebuggerPresent	124 (4.33)	1 (0.11)	432 (0.95)	◐		01/10	09/19	9	24	2	N
	[14]	GetAdaptersAddresses	58 (2.02)	4 (0.42)	293 (0.65)	◑	◇	02/11	09/19	46	135	13	N
	[44]	GlobalMemoryStatusEx	78 (2.72)	26.0 (2.74)	250 (0.55)	◑	◇	11/10	09/19	5	32	1	N
		NtQuerySystemInformation(0x23)	28 (0.98)	0 (0)	49 (0.11)	◐		03/10	09/19	10	33	3	N
		DeviceIoControl	2 (0.07)	2 (0.21)	7 (0.02)	◑		07/13	11/17	14	62	4	N
		PEB→NumProcessors	7 (0.24)	0 (0)	11 (0.02)	◑		03/12	11/17	1	13	0	Y
		NtQueryObject after NtCreateDebugObject	0 (0)	0 (0)	0 (0)	◐		Never	Never	21	93	6	N
WMI	[51]	ExecQuery	0 (0)	0 (0)	0 (0)	◑		Never	Never	25	114	10	N

Table 4: Techniques **D/V**: ◐Anti-Debugging, ◑Anti-VM, ●Both. ◇Used by Goodware. **BB** is the number of basic blocks, and **I** is the number of instructions, **C** is the number function calls, **ASM** indicates whether implementing the given technique requires writing assembly code.

		Technique	#Families (%)	#Goodware (%)	#Malware (%)	D/V	G.	First	Last	BB	I	C	ASM
Process Environment	[41] [37] [39]	NtSetContextThread(CONTEXT_DEBUG_REGISTERS)	257 (8.96)	0 (0)	1617 (3.56)	◐		01/10	09/19	13	40	2	N
		NtQueryInformationProcess(0x07)	280 (9.77)	4.0 (0.42)	1028 (2.27)	◐		01/10	09/19	13	42	4	N
		NtSetInformationThread(0x11)	122 (4.26)	1.0 (0.11)	350 (0.77)	◐		01/10	07/19	14	46	7	N
		NtQueryInformationProcess(0x1e)	96 (3.35)	0 (0)	303 (0.67)	◐		01/10	08/19	15	56	6	N
		NtQueryInformationProcess(0x1f)	34 (1.19)	0 (0)	115 (0.25)	◐		10/12	08/19	13	41	4	N
		NtGetContextThread(CONTEXT_DEBUG_REGISTERS)	28 (0.98)	0 (0)	46 (0.1)	◐		01/10	03/18	14	42	2	N
		DebugActiveProcess on Parent	2 (0.11)	0 (0)	2 (0.01)	◐		10/14	02/16	54	154	21	N
		NtCreateThreadEx(HIDE_FROM_DEBUGGER)	0 (0)	0 (0)	0 (0)	◐		Never	Never	16	50	6	N
File System	[49]	NtOpenFile	613 (21.38)	6.0 (0.63)	3268 (7.2)	●	◇	01/10	09/19	7	40	2	N
		NtQueryAttributesFile	536 (18.7)	14.0 (1.48)	2616 (5.77)	●	◇	01/10	09/19	7	34	2	N
		NtCreateFile	389 (13.57)	6.0 (0.63)	1696 (3.74)	●	◇	01/10	09/19	9	49	2	N
		NtQueryDirectoryFileEx	0 (0)	0 (0)	0 (0)	●		Never	Never	81	330	23	N
List Proc.	[49]	NtQSI(SYSTEM_PROCESS_INFO)	649 (22.64)	29.0 (3.06)	4054 (8.93)	●	◇	01/10	09/19	47	189	22	N
		EnumProcesses	122 (4.26)	11.0 (1.16)	393 (0.87)	●	◇	02/10	12/18	58	201	16	N
		GetModuleBaseName	0 (0)	0 (0)	0 (0)	●		Never	Never	72	254	25	N
List Services	[49]	OpenSCManager	222 (7.74)	2.0 (0.21)	1029 (2.27)	◑		01/10	09/19	82	340	27	N
		EnumServicesStatus	36 (1.26)	0 (0)	167 (0.37)	◑		01/10	09/19	82	340	27	N
		OpenService	6 (0.21)	0 (0)	6 (0.01)	◑		02/19	08/19	39	167	20	N
		GetServiceDisplayName	0 (0)	0 (0)	0 (0)	◑		Never	Never	24	87	10	N
		GetServiceKeyName	0 (0)	0 (0)	0 (0)	◑		Never	Never	24	87	10	N
Driv	[52]	EnumDeviceDrivers	5 (0.17)	0 (0)	21 (0.05)	◑		01/15	08/18	60	198	16	N
		GetDeviceDriverBaseName	4 (0.14)	0 (0)	7 (0.02)	◑		01/15	10/16	60	198	16	N

and we never observed the use of WMI.

Keywords Blacklist. Some techniques make access to system resources such as files, directories, Windows registries, processes, and services. Malware samples actively search for analysis and monitoring tools to evade detection. In fact, the presence of such tools is a symptom of being on an analyst's machine. Besides keywords related to virtual machines and debuggers, we inserted keywords relative to these analysis tools (Anti-Analysis). Table 5 shows the blacklist defined for the analysis. The Anti-VM group contains some keywords related to common Virtual Environments used for the analysis. Tools considered are: Xen Hypervisors, VirtualBox, VMware, Parallels, Wine, QEMU, Cuckoo, and Bochs. The blacklist has been built selecting keywords related to VMs, debuggers, and malware analysis tools. Some of them have been collected from threat reports [40, 53], while we found others inside our dataset. These resources should not be visible to evasive malware because it could use them to identify our system. Indeed, we deny access to all sensible resources.

Packing. Commercial packers often employ anti-debugging techniques. To explore such a phenomenon and validate our evasion detection techniques, we ran PEiD [54] on the samples in our dataset. PEiD detected packers in 4,293 samples, with the most common packers being UPX (1713), Nullsoft PiMP (1609), PECompact (237), ASPack (185). Furthermore, PEiD detected 5

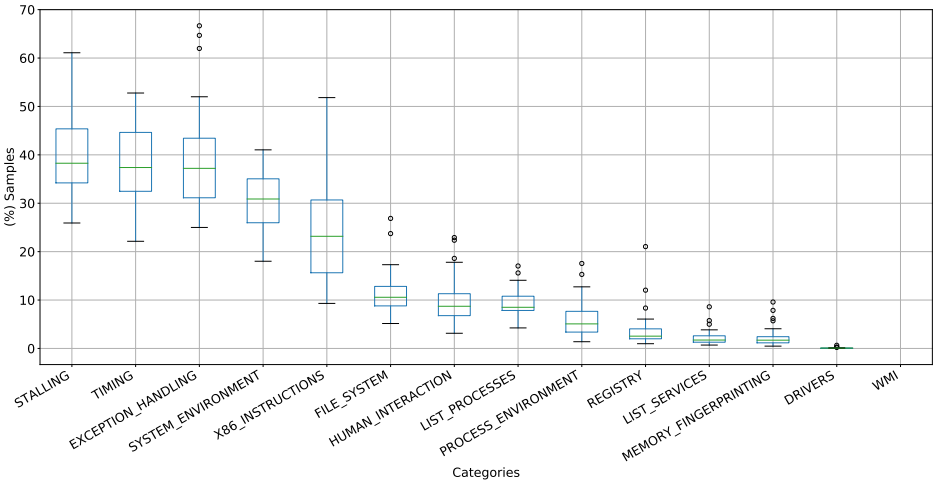


Figure 2: Percentage of evasive samples by category.

samples packed with (different versions of) Themida, which is known to adopt anti-debugging techniques. For all the Themida samples, our framework detected Timing, Exception Handling, Traps, and Registry techniques. Interestingly, for versions < 1.8 we identified 3 techniques: `GetLocalTime` (Timing), `NtClose(INVALID_HANDLE)` (Exception Handling), ANTI-VM `VPCEXT` (Traps). In later versions, we additionally found `RDTSC/D` (Timing) and `NtOpenKey/Ex` (Registry).

It is important to note, the amount of packers that we identified is a *lower bound*. Indeed, PEiD is able to identify just common packers, but it does not run an entropy analysis. This implies that custom and not very widespread packers were not identified.

Techniques Clusters. We analyzed the use of techniques in malware samples by computing the Pearson correlation coefficient between each technique. We then created clusters of techniques whose correlation was above 0.5 (highly correlated). Table 6 show the cluster of techniques that are often employed in the same malware samples.

RQ1. *The vast majority of malicious samples adopt at least one evasive techinque, with Timing and Stalling being the most adopted categories of techniques. Malware families also look for artifact keywords, with "vbox", "sandbox", and "virtualbox" being the most used keywords.*

Table 5: Blacklist of functions accessing OS resources

Anti-VM
vbox, virtualbox, vmware, vmx, vmsrvc_, prl, xen, parallels, bochs, wine, qemu, vmusrvc_, innotek, vmhgfs
Anti-Debugging
ollydbg, debugger, debug, softice, x32dbg windbg, immunity debugger, dbgviewclass
Anti-Analysis
wireshark, tcpview, autoruns, regmon, procexp, hookexplorer, sysinspector, petools, dumppcap, python, malware, virus, sample, anyprotect, process viewer, bitdefender, seguridad, charles, dll injector, fiddler, file monitor, filemon obsidiangui, processhacker, procmon, progman regmon, rock debugger, rootkitrevealer, socketsniff, pe explorer, process explorer, process hacker, process heap viewer, process monitor, program manager, packet analyzer, registry editor, resource hacker, security task manager, switchsniffer, tcpview, wireshark, windows file protection, winhex, taskmgr, trw2000, remote process viewer, httpanalyzer, mcafee, tidawindow, smartsniff, kaspersky, norton, panda, webroot, sophos, avira, trendmicro, comodo, avira, bitdefender, clamav, symantec, avast, nod32, drweb fortinet, g data, systracer, regshot, windows task manager

Table 6: Techniques that are employed in the same malware samples.

`NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)` `GetTickCount`, `RDTSC/D`, `CPUID(eax=0x00000001)`, `waitForSingleObject/Ex` `QueryPerformanceCounter`, `SetUnhandledExceptionFilter`, `Sleep/SleepEx`
`SGDT`, `SLDT`, `SIDT`
`IN`, `VPCEXT`
`NtQueryInformationProcess(0x1e - 0x1f - 0x7)`, `POP SS` `CheckRemoteDebuggerPresent`, `NtSetInformationThread(0x11)`
`GetModuleName`, `EnumProcesses`
`STR`, `POPFD`

Table 7: Usage of blacklisted keyword per families in our dataset

Family	keyword	#
neshta	sandbox	444
	virtualbox	443
	xen, sample	442
	python	424
	debug	353
	vbox	226
	malware	203
outbrowse	vbox	302
installmonster	virtualbox	129
sivis	debugger	127
hotbar	debug	118
simda	wireshark, virtualbox, debug, vbox	85

5.3. Families & Evasive Techniques

Intuitively, the set of evasive techniques used and the malware families could be correlated. This hypothesis is raised from the way malware samples are developed. In fact, malware samples are usually built by tools. The production chain of malware has been proved to be well distinct. Malware authors usually develop builders that can be easily used to generate new samples. The malware authors benefit from the selling of the build because the criminals who distribute malware are not necessarily the ones who wrote it or even able to develop a sophisticated malware program. Evasive techniques are usually an additional feature of malware sample builders. We leveraged a machine learning algorithm of pattern identification to evaluate this correlation between the evasive techniques and malware families. We set up a RandomForest Classifier; the input of the classifier is an array of boolean variables representing the use of each technique inside a sample; the output of the classifier is the family label. We built 123 classifiers for 123 families (all families with at least 50 samples). The output of each classifier is trained with 3-fold cross-validation using stratified sampling. For each classifier, we computed the f1-score, a metric to evaluate how well the classifier is performing, taking all the confusion matrix into account. In Figure 3, we

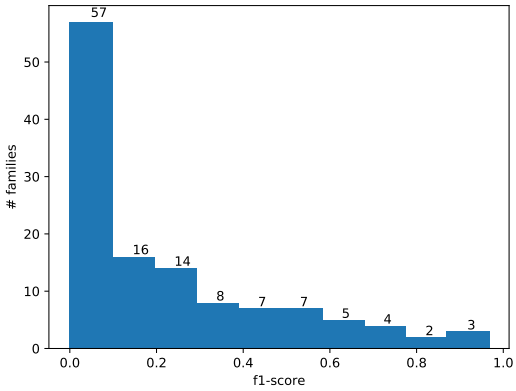


Figure 3: F1-score for RandomForestClassifier; y-axis shows the number of correctly classified family, x-axis is ordered by f1-score.

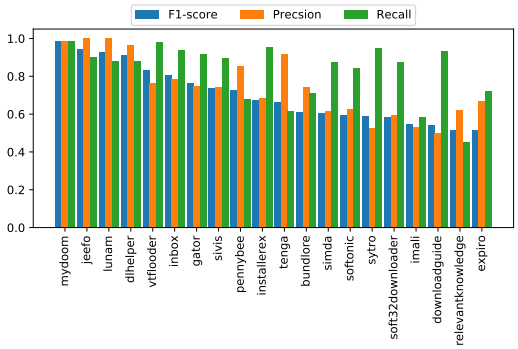


Figure 4: x-axis shows the name of the family that was used to build the classifier. y-axis show the scores for precision, recall and f1

can appreciate the result of this experiment. While clearly, there are families with a solid correlation, the values of the F1-score drop quickly. This implies that from our analysis, 16 families can be recognized from the combination of evasive techniques employed. Indeed, F1-score is not enough to evaluate the performance of a classifier. An F1-score greater than 0.5 can still be produced by a low precision and recall. Figure 4 show more measurement about the performance of such classifiers. From Figure 4 we can see that two of those classifiers (imali, relevantknowledge) have low precision and recall.

Malware samples are often packed. Some packers, like obsidium, can implement evasion techniques themselves. Thus, we ran an additional experiment to identify if there is any correlation between the packer that we identified in our dataset and the techniques. The experiment setup is similar to the previous one. In this case, instead of building a classifier for the family, we built a classifier for packers. We lowered the minimum threshold of samples to 20 to include more packers. In total, we classified 33 packers; the results are shown in Figure 5. Only 'PEtite' showed a significant correlation between used techniques and the packer (F1: 0.87, Recall: 1.0, Precision: 0.82).

Finally, while we considered tf-idf as techniques to prove correlation among techniques and families, we chose Random Forest to study further if a detector with good performance could be built for some of the families and packers.

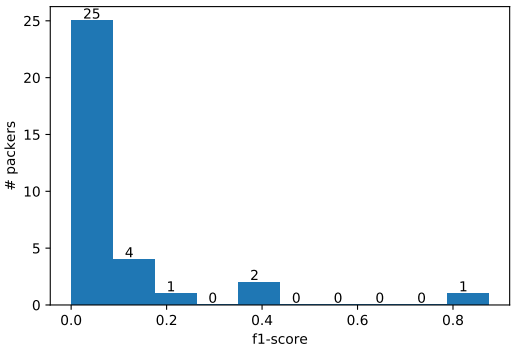


Figure 5: F1-score for RandomForestClassifier; y-axis shows the number of correctly classified packers, x-axis is ordered by F1-score. The only packer with F-1 score greater than 0.5 is `PEtite`

Figure 6: Percentage of evasive samples per quarter. Straight lines are linear regressions: In orange for evasive techniques, in blue for packers.

RQ2. There is a consistent number of malware families (46) that are characterized by the selection of evasive techniques they employ.

5.4. Longitudinal Analysis

Our dataset has been collected evenly spread across ten years (Section 5.1). This dataset empowered us to perform analysis and identify changing trends during these years.

Interestingly, Figure 6 shows that there was a little increment ($\sim$ 12%) in the use of evasive techniques during the last ten years. In comparison, the use of common packers stayed constant. The detection of all techniques starts from 67.8% in 2010 and slightly increases to 80% during the last years. This result is coherent with old works[27, 30]. Although, in Section 5.6, we discussed how several techniques could be confused with normal program behavior.

Moreover, we analyzed the intricacy of malware during the last ten years. Figure 7 shows the maximum number of evasion techniques that were used by a single sample. It is interesting to notice the rising of the number of techniques until 2017. However, we can see a decrease in the last two years.

Furthermore, we analyzed the targets of evasive attempts. Indeed, our system can discern when the malware is evading a debugger or a virtual machine. In Figure 8 we show four categories of targets.

We exploited the categorization in Tables 3-4. However, for techniques used for both Virtual Machines and Debuggers, we relied on the blacklist de-

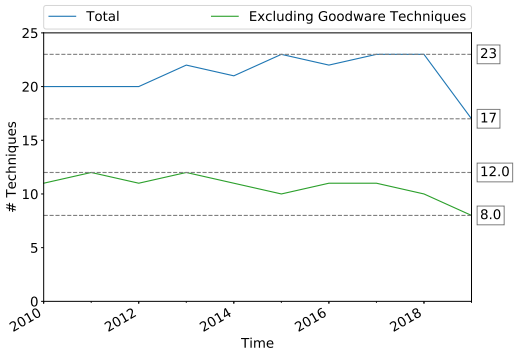


Figure 7: The maximum number of evasive techniques detected in a single sample, yearly aggregated.

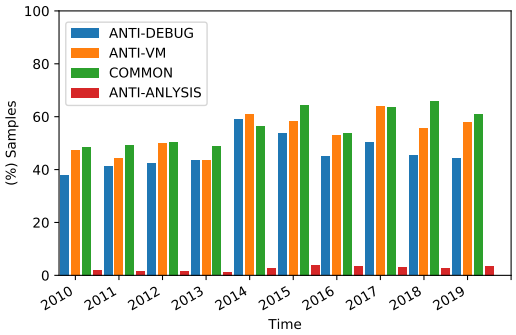


Figure 8: Percentage of samples divided by targets.

scribed in Section 5.2. There are few techniques categories (Stalling, Timing, Process Enumeration) that cannot be assigned to either of the categories. We named these techniques *Common*. Figure 9 shows how the type of techniques used by malware authors shifted during the last 10 years.

Our blacklist includes, besides virtual machine and debugger, generic analysis tools keywords. In fact, finding classical analysis tools is a clue that the machine is used for analysis. The *Anti-Analysis* peak in 2018 is caused by a campaign of the `neshta` family that scans the disk searching for files to infect. In 2015 instead, the predominant *Anti-Analysis* families were `simbda` looking for `wireshark` and `eorezo` searching for antivirus software. Keywords used for most common families are shown in Table 7.

Another interesting aspect is the timeline of the techniques: lots of them have been observed during the whole timeframe considered, which means that their first appearance may go back to years before the ones considered in this work and that malware authors will probably continue to use them in the future. Figure 10 shows the period of utilization of each technique. We restricted the graph to techniques that are predominantly found in malware (See Section 5.6). The graph is based on the observation of running samples, and the dates refer to the first appearance of the corresponding sample on Virustotal. Several techniques start at the beginning of 2010. Very likely, those techniques have been used before 2010. The same reasoning can apply to the ending: the closer the ending date is to the end of our dataset (September 2019) more likely it is that the technique is going to be used in the future. We plotted each observation — of an evasive technique —

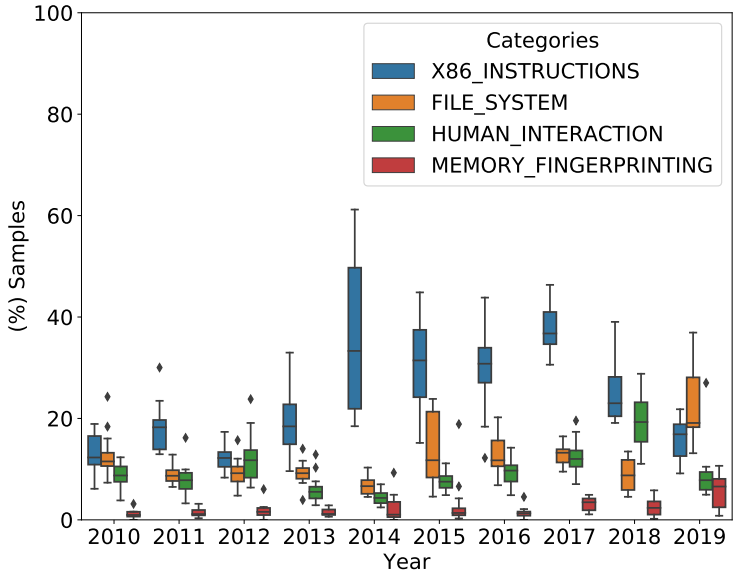
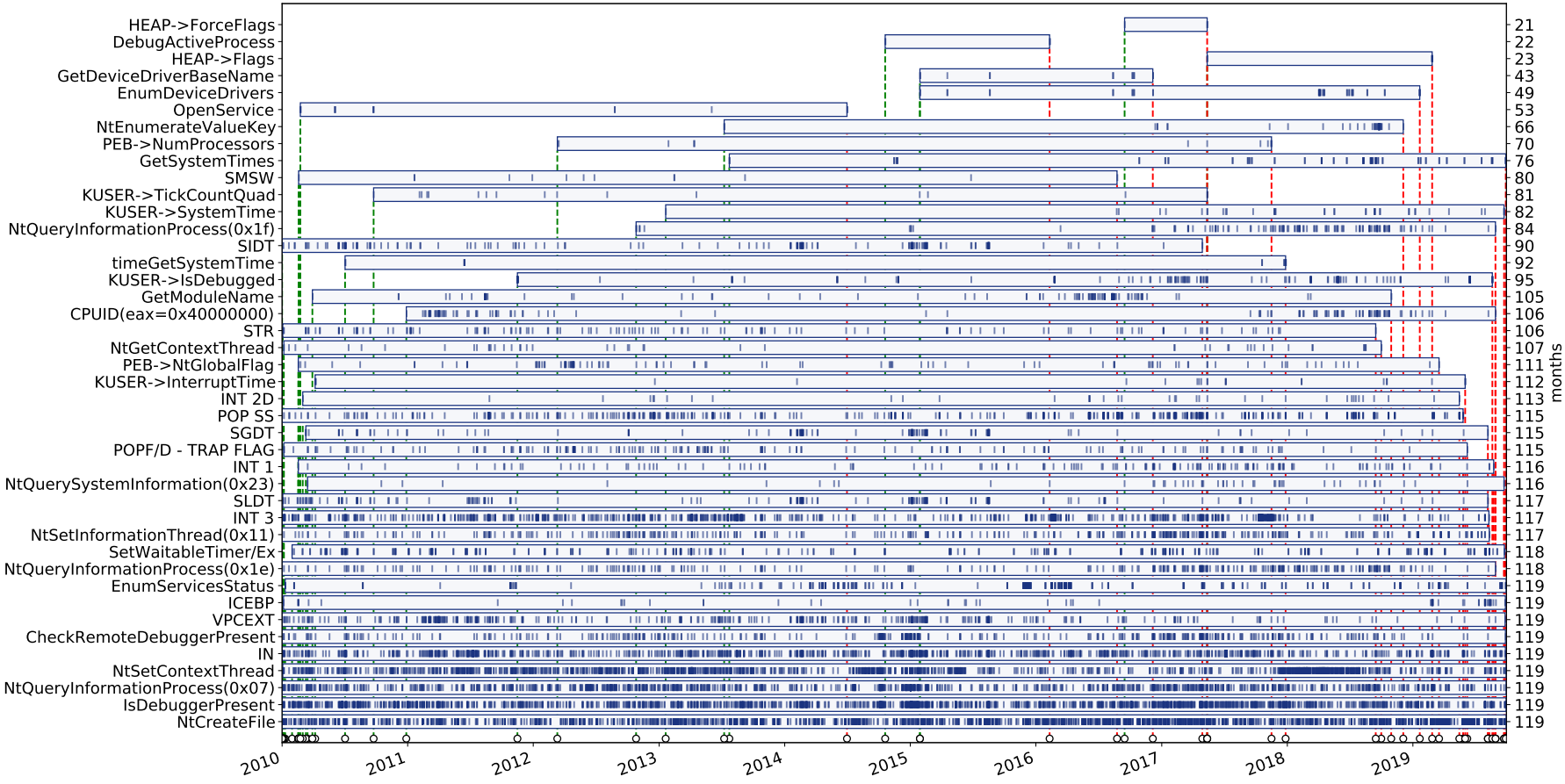


Figure 9: Timeline of techniques usage.

as a vertical line on the corresponding bar. We can notice two patterns: (1) many techniques shows cluster in small periods. This can be attributed to active malware campaigns, either because analysts submit newly found samples or because malware authors test their samples before release. (2) Some techniques are more used than others. We can notice like both `INT 2D` and `IsDebuggerPresent` are really old, but the latter is definitely more used. Indeed, `IsDebuggerPresent` is documented by Microsoft API documentation and technically easier to implement; these may have contributed to its popularity as an evasion technique.

Furthermore, we can notice techniques that are not used anymore, like `PEB->NumProcessors` that checks the number of cores and `SIDT` that exploits a side-effect of virtualization techniques in old CPUs. Those techniques are not effective anymore since the rise of the number of cores in modern processors allowed virtual machines to use more than a single core and modern CPUs support for virtualization removed some of the side-effects.

Figure 10: Time-line of techniques usage in malware samples. Each horizontal line indicates the period in which the technique has been observed. Each blue tick is one observation of a sample using the technique.



RQ3. *Our longitudinal study highlights that evasive techniques were already quite common back in 2010, with almost 70% of malware samples employing evasive behaviors, and their adoption slightly increased over the past ten years, although not significantly. Moreover, we can see malware authors drop techniques that are not effective anymore.*

5.5. Evasion vs. Community

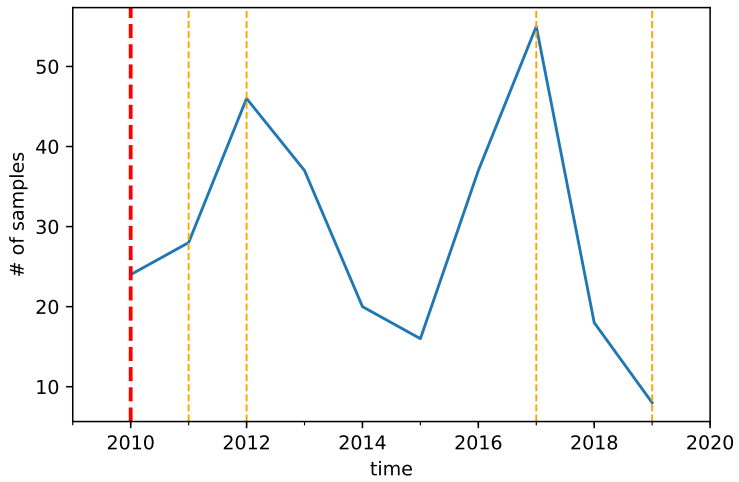


Figure 11: `POP SS` detection timeline. The blue line is the number of samples that adopt `POP SS`. The red vertical line is the first public appearance of the technique, the orange lines any sequent appearance.

Evasive techniques have been reported for years in research papers, industrial conferences, and specialized forums. In this section, we conduct a preliminary study on the impact of the adoption of malware evasion techniques on the security community and vice versa. In particular, we aim at estimating (1) the influence of the release of information about evasive malware techniques on their adoption and (2) the time required by the community to report a new evasive technique with respect to its first appearance in our dataset.

We crawled the Internet for information related to the evasive techniques under analysis to estimate the community awareness on malware evasiveness. We focused on academic and industrial works scraping paper, blog posts, and technical reports. We searched for the keywords related to the techniques listed in Table 3 and Table 4. Then, we filtered the results associated with the concept of malware or evasion (e.g., "IsDebuggerPresent" AND (malware

OR evasive))[1]. We further post-processed the gathered data by manually analyzing and filtering out non-relevant results (i.e., generic reports of dynamic analyses with no explicit references to evasive behaviors). Coherently with the longitudinal study conducted in this paper, we focused on the period from 2010 to 2019. Additionally, we crawled the same information for the period before 2010 to study the impact of the security community on already known techniques. We collected a total of 394 sources[2]. Then, we compare each evasion technique's quarterly trends (i.e., the number of sample implementing it) with the harvested information. Despite this study provides only an estimation of the influence of the community on the adoption of evasive techniques – and vice versa – since it might be biased by both the dataset under analysis and the (publicly available) sources, we deem that it provides insights for future research on the topic. In fact, in the period under analysis, we observe an increasing trend in the number of resources related to malware evasiveness: between 2010 and 2015, we have an average of 3 reports per year, while between 2016 and 2019, this number doubled, demonstrating the higher community interest and awareness of the problem, passing the years. From the 92 evasive techniques under analysis, 50 were already known before 2010, while 21 are disclosed by the community between 2010 and 2020. On average, techniques become public after 4.4 years. The fastest disclosure belongs to the 'EnumDeviceDrivers' technique, which appears in malware samples in the same quarter of the talk [55]. Instead, the slowest disclosure— after 7 years—belongs to 'VPCEXT' [56].

By comparing the quarterly trends of each evasion technique with the harvested information, as shown in Figure 11 for the `POP SS` technique, it is possible to observe an evident influence between them. There is a high correlation between the release of information about evasive techniques and their implementation in malware. In particular, if the technique is already widely adopted/known, after the release of new information, we notice a slight and temporary decrease in their usage in the subsequent years. On the contrary, if the technique is scarcely used/known, we observe a swift increase in its adoption in the following years after the release of reports. As expected, once a technique becomes widely used, the number of related

[1]To limit the bias of search engines' personalized search and geolocalization, we worked in "Incognito mode".

[2]The gathered data is available at https://github.com/necst/brioscia

reports significantly increases.

RQ4. *Our analysis highlights a correlation between the adoption of malware evasive techniques and the information provided by the security community: if from one side, we observed that the release of security reports on evasive techniques corresponds to an increase of their adoption in the short term, on the other, it tunes down their spread on the long term.*

5.6. Malware vs. Goodware

While less common, goodware also employs evasion techniques to prevent reversing and protect intellectual property (e.g., games, Spotify). Moreover, some of the evasive techniques that we monitor may be based on side-effects and operations that are not intrinsically malicious, and that can be used both for evasive and non-evasive behaviors. For instance, Operamail monitors the list of running processes to look for another instance of itself, and it does not run if another instance is already running.

In order to better characterize such techniques, we collected a dataset of benign applications. We discarded all those binaries that were positively detected by any anti-malware used by VirusTotal. The goal is to distinguish techniques that are used exclusively for evasion purposes from the ones that are, instead, often used by legitimate programs for benign behaviors. For instance, `CPUID(eax=0x1)` is an instruction that Microsoft compiler adds at the beginning of any binary. While this instruction can be used to fingerprint the CPU and be used to understand if the program is executed inside a virtual machine, it can also be used to understand if a library needs to use a software implementation of floating-point arithmetics or it can exploit the hardware FPU.

We then analyzed these applications to identify evasive patterns. Figure 12 shows the number of evasive techniques adopted by legitimate programs in our goodware dataset (30). We can see that some evasive patterns are present in many goodware programs. Sometimes, goodware implement anti-analysis techniques to protect Intellectual Property. For this reason, we manually analyzed the use of 18 techniques. We choose to analyze the techniques that are less frequently used in benign programs. One (`IsDebuggerPresent`) of these eighteen techniques were used as an evasion technique to identify the presence of a debugger. The remaining 17 were not used as an evasion mechanism. We classified each technique based on our goodware analysis:

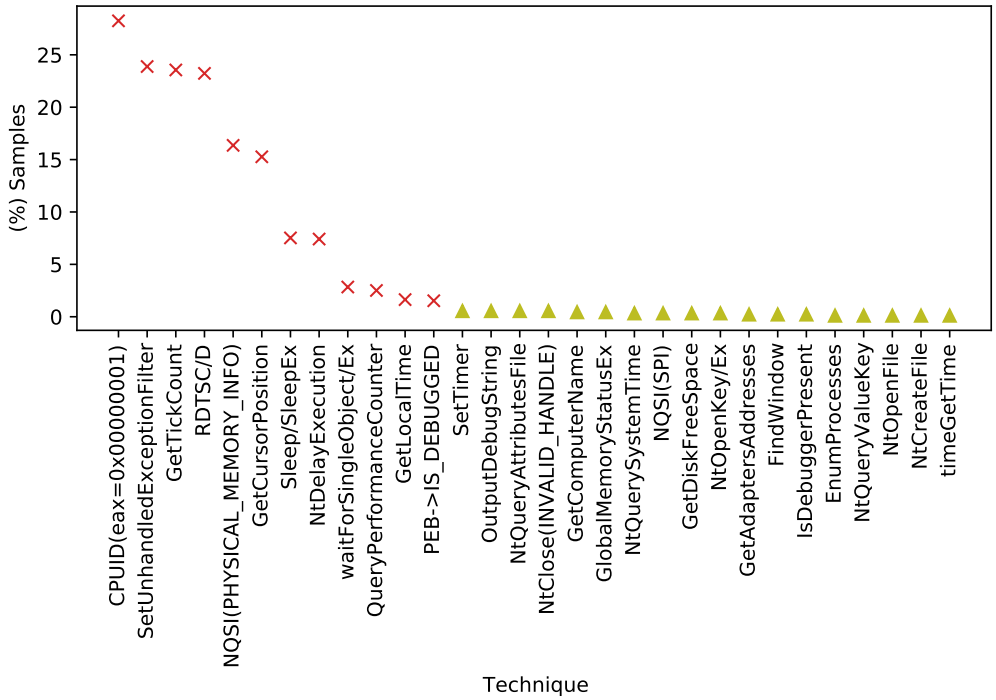


Figure 12: Appearance of evasive techniques in Goodware dataset. × markers identify unreliable techniques because of their massive presence in goodware programs. △ markers identify techniques that we have manually analyzed in goodware programs to determine if the technique is used for evasion.

Used by Malware. It is the evasive behavior never observed inside goodware samples, or we manually verified that it is used as an evasion mechanism in goodware samples.

Used by Goodware. It is the evasive behavior observed in more than 1% of our goodware samples, or we manually verified that there were not used as an evasive behavior.

If we consider this division of techniques and revisit some of the results we have gotten so far, we can notice some differences. Indeed, looking at Figure 13, we can see that if we consider only malware-related techniques, the percentages of evasive families drop from 60-85% to 19-30%. While the use of techniques used by goodware is increasing, the use of techniques that are explicitly only in malware stays constant. Similarly, looking at Figure 7, if we consider all techniques, there is an increase in usage until 2017. If we take into account only techniques found exclusively in malware, the trend

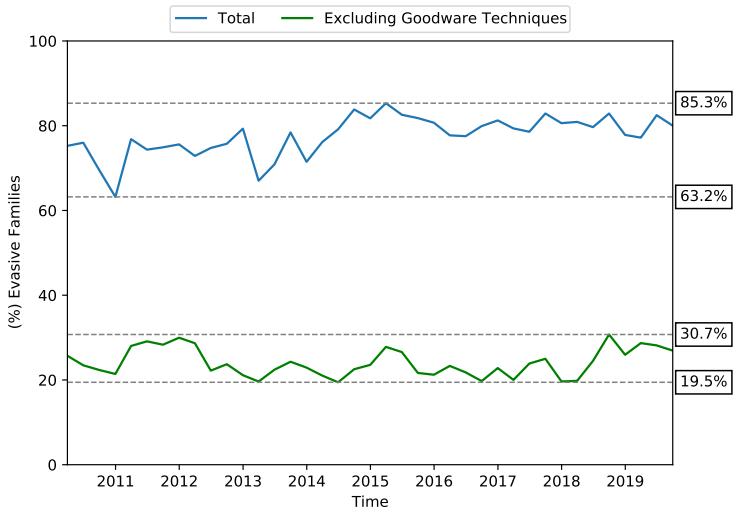


Figure 13: Ratio of evasive families quarterly aggregated.

shows a constant decrease in the number of techniques used in the same sample. We also remark that techniques never detected in benign programs are the ones used explicitly for malicious purposes. Even though we validate our benign dataset against VirusTotal to minimize the presence of trojan in our benign dataset, it is possible that some Trojan was not caught by our validation. The lower bound results may be influenced by those trojans.

Techniques Clusters Similarly, for what we did for malware techniques, we computed the Pearson correlation among techniques that are implemented in the same goodware. Techniques with correlation > 0.5 are shown in Table 8. Interestingly the first group shares all but one (`GetCursorPosition`) technique with the first group of malware samples (Table 6). Besides that, all other clusters are different from the ones seen for malware samples.

RQ5. *Legitimate programs adopt evasive techniques, even if this phenomenon is significantly less common. In particular, only a few techniques are commonly adopted, suggesting that such techniques are not mainly used for true evasive purposes.*

6. Limitations

Reliable detection of evasive techniques is a complex engineering problem, as a "perfect" solution often does not exist, and the adversarial context exacerbates the issue. We acknowledge that our analysis framework itself

Table 8: Techniques that are employed in the same goodware sample.

GetCursorPosition, RDTSC/D, GetTickCount, SetUnhandledExceptionFilter, NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)
NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION), GetAdaptersAddresses, EnumProcesses, timeGetTime
IsDebuggerPresent, NtQueryValueKey, NtOpenKey/Ex, GetDiskFreeSpace
NtCreateFile, NtOpenFile
NtDelayExecution, Sleep/SleepEx
NtQuerySystemTime, GetComputerName, NtClose(INVALID_HANDLE)
OutputDebugString, PEB→IsDebugged

can be evaded: We measure the usage of the 92 (known) techniques that our framework supports, but of course, samples may employ unknown techniques that we are unable to track.

Our choice of selecting samples detected by at least 35 AV engines allows us to exclude false positives (i.e., legitimate programs that we would consider malicious). However, as a side effect, our dataset may not include true malicious samples that are not well detected by the majority of AV engines. Unfortunately, this is an inevitable trade-off since manual verification of each sample is not feasible when performing large-scale studies such as ours. We plan to investigate evasive behaviors in less detected samples in future research.

Similarly, to avoid malicious samples into the benign dataset, we filtered out any binary that was detected as positive by an AV on VirusTotal. Unfortunately, we cannot be certain that our dataset does not contain any Trojan.

Coherently with the research aim to perform a longitudinal analysis, during data collection, we cared about temporal distribution of the number of samples trying to sample the same amount of binaries each month. Of course, this affects the distribution of samples per family. As we can see in Table 2, even in the top 20 most common families in our dataset, there is a significant difference in the number of samples. In fact, the type of families in a month depends on the currently active malware campaigns and samples submitted to VT. Moreover, this also depends on the type of family itself. In fact, if the malware sample is polymorphic or metamorphic, the number of samples is very high. Instead, for non-morphic malware, we see only one sample. This

implies that when we look at results by the number of samples, the results may be affected by the most prominent families. For these reasons, we also provide, when possible, results based on the number of families.

Some technical limitations of our current implementation stem from the use of a DBI framework for the analyzer. The stability of the injection and the performance impact of the DBI tool are significant challenges. Also, the artifacts induced by the DBI tool itself could be exploited by new evasive techniques [18]. This means that our results might underestimate malware evasiveness.

Moreover, any evasion attempt that is performed after 5 minutes is lost by our framework. If we consider the most evasive families (>95%) and we assume that those samples were misclassified by our framework, our framework lost evasion techniques for at least 121 samples. Although, accordingly with Küchler et al. [35], most malware samples execute within 2 minutes timeframe. This implies that the execution time should not have a significant effect on our results.

The techniques we monitor are indicators of potential evasion attempts, but they do not capture the intention of malware authors. We can only speculate why malware authors are adopting the techniques that we identify in our study.

7. Discussion and Future Directions

In this paper, we shed light on the evasive behaviors employed by modern Windows malware by systematically documenting and classifying 92 evasion techniques and by measuring their adoption over the past 10 years.

From a longitudinal, historical perspective, our results show that, overall, the evasion rate had a small increase over the years, starting very high since 2010 (when it was already almost 70%) and reaching 80% at the end of 2019. We also documented a decrease in the intricacy of malicious techniques over the last years, with Timing techniques increasingly becoming their first choice. We also found an interesting relationship with publications of techniques by the research and anti-malware community, with an immediate increase in adoption, followed by a steeper decline.

While we can only speculate the reason behind the malware authors' choices, we believe that the main reason for the observed increase in the adoption of evasive behaviors is the fact that the studied techniques are still quite effective for the cybercriminals' purposes (i.e., evading dynamic

analysis and delaying the characterization of their samples). In spite of the considerable amount of research proposed in the past 10 years, mitigation of evasive malware in the real world is still a problem. While, on the one hand, designing an "invisible" dynamic analysis environment is an extremely challenging task, which can be linked to the halting problem, on the other hand, we believe that the security community can and should research novel and practical solutions to mitigate this problem further. In particular, while online sandboxes nowadays implement many anti-evasion strategies, not many endpoint anti-malware solutions employ dynamic-analysis approaches [10]. Those that do adopt lightweight solutions are, unfortunately, incompatible with the resource-hungry anti-evasion approaches currently available. To overcome this, vendors often send suspicious executables to cloud-based engines for more complex analyses, but this leaves open a window of time for the execution of malicious payloads on endpoints, which in the case of ransomware may be enough to make the subsequent cloud-enabled detection pointless. Thus, evasive behaviors allow malware authors to "delay" the identification of their samples to a point when they already caused substantial damage. We believe that future research should focus on re-designing approaches that mitigate evasive malware to make such solutions practical and efficient so that they can be deployed inside emulators that run directly on the endpoints, allowing for faster detection.

Our study also found that malware authors move from old techniques to new ones, pursuing their goal to evade a dynamic analysis system. Therefore, it is essential for forward-looking anti-evasion solutions to be able to capture the fundamental properties of evasive behavior and to adapt to emerging techniques quickly.

Moreover, our analysis framework—similarly to the previously proposed anti-evasion tools—currently does not identify the adoption of logic bombs for evasion purposes [57], which we plan to study in future work.

An additionally interesting future work is the analysis of evasive techniques targeting the 64-bit x86 architecture. Our analysis in this paper is based on binaries using the 32-bit x86 instruction set architecture because it is the architecture prevalently used by malware authors nowadays. However, there are a few differences in its 64-bit counterpart that may change how some of the evasive techniques work.

Finally, no previous study focused on evaluating the effectiveness of the currently deployed anti-evasion solutions. In fact, some existing and old techniques are currently still being adopted, which suggests that they are

still effective. We believe that future work should research novel metrics and principled approaches to test and compare sandboxes and instrumentation frameworks, ultimately assessing and quantifying their fingerprintability and robustness to evasiveness.

8. Conclusions

In this paper, we systematically documented 92 evasion techniques (to our knowledge, the broadest collection so far in literature) adopted by modern Windows malware and created a meaningful taxonomy of them. We implemented and released a DBI-based tool to analyze Windows executables and identify whether they employ any of such evasive techniques. Leveraging our analysis tool and a dataset of 45,375 malware samples from 2,867 different families observed in the wild over a span of 10 years, we performed a measurement of the evasion phenomenon over time. We showed that the adoption of evasive techniques slightly increased over the years and is nowadays common in modern malware and that malware authors update their techniques to evade analysis systems. From a non-temporal perspective, we found a high correlation between sets of malware families and associated evasion techniques. We also identified 15 published techniques that do not seem to be prevalent in our dataset: we will investigate this result more in-depth in future extensions of this work. Finally, we analyzed the prevalence of (some) evasive behaviors in legitimate executables by running a comparative analysis on a dataset of goodware samples. This is the first work that shows a large collection of evasive behaviors and a quantitative analysis of their distribution over time. We believe it provides interesting insights to explore other methodologies to detect novel classes of evasive behaviors.

Acknowledgements

We would like to thank our reviewers for their valuable comments and input to improve our paper. We would also like to thank VirusTotal for providing us access to some of malware samples used for this project. This work was partially supported by research funding provided by BVTech SpA.

References

[1] A. Continella, A. Guagnelli, G. Zingaro, G. D. Pasquale, A. Barenghi, S. Zanero, F. Maggi, Shieldfs: a self-healing, ransomware-aware filesys-

tem, in: S. Schwab, W. K. Robertson, D. Balzarotti (Eds.), Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016, ACM, 2016, pp. 336–347.
URL http://dl.acm.org/citation.cfm?id=2991110

[2] A. Kharraz, W. Robertson, D. Balzarotti, L. Bilge, E. Kirda, Cutting the gordian knot: A look under the hood of ransomware attacks, in: Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), Springer, 2015.

[3] A. Continella, M. Carminati, M. Polino, A. Lanzi, S. Zanero, F. Maggi, Prometheus: Analyzing webinject-based information stealers, Journal of Computer Security 25 (2) (2017) 117–137. doi:10.3233/JCS-15773.
URL https://doi.org/10.3233/JCS-15773

[4] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, G. Vigna, Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS), 2018.

[5] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, M. Bailey, Outguard: Detecting in-browser covert cryptocurrency mining in the wild, in: In Proceedings of the The World Wide Web Conference (WWW), 2019.

[6] R. Holz, D. Perino, M. Varvello, J. Amann, A. Continella, N. Evans, I. Leontiadis, C. Natoli, Q. Scheitle, A retrospective analysis of user exposure to (illicit) cryptocurrency mining on the web, in: Proceedings of the Network Traffic Measurement and Analysis Conference (TMA), 2020.

[7] M. Labs, Mcafee labs threats report, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf (Jun. 2017).

[8] McAfee.com, Mcafee labs threats report, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf (Dec. 2018).

[9] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, P. G. Bringas, Sok: deep packer inspection: a longitudinal study of the complexity of run-time packers, in: Proc. of IEEE symposium on Security and Privacy (SP), IEEE, 2015.

[10] D. Quarta, F. Salvioni, A. Continella, S. Zanero, Toward systematically exploring antivirus engines, in: Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2018.

[11] T. Raffetseder, C. Kruegel, E. Kirda, Detecting system emulators, in: International Conference on Information Security, Springer, 2007, pp. 1–18.

[12] R. Paleari, L. Martignoni, G. F. Roglia, D. Bruschi, A fistful of redpills: How to automatically generate procedures to detect cpu emulators, in: Proceedings of the USENIX Workshop on Offensive Technologies (WOOT), Vol. 41, 2009, p. 86.

[13] J. Rutkowska, Redpill: Detect vmm using (almost) one cpu instruction, http://invisiblethings. org/papers/redpill. html (2004).

[14] M. Lindorfer, C. Kolbitsch, P. Milani Comparetti, Detecting environment-sensitive malware, in: R. Sommer, D. Balzarotti, G. Maier (Eds.), Recent Advances in Intrusion Detection, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 338–357.

[15] N. Miramirkhani, M. P. Appini, N. Nikiforakis, M. Polychronakis, Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts, in: Proceedings of the IEEE Symposium on Security and Privacy (S&P), 2017.

[16] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, S. Zanero, Identifying dormant functionality in malware programs, in: 2010 IEEE Symposium on Security and Privacy, 2010, pp. 61–76.

[17] D. Kirat, G. Vigna, C. Kruegel, Barecloud: Bare-metal analysis-based evasive malware detection, in: Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 287–301.

[18] M. Polino, A. Continella, S. Mariani, S. D'Alessio, L. Fontana, F. Gritti, S. Zanero, Measuring and defeating anti-instrumentation-equipped malware, in: M. Polychronakis, M. Meier (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, 2017, pp. 73–96.

[19] D. C. D'Elia, E. Coppa, F. Palmaro, L. Cavallaro, On the dissection of evasive malware, IEEE Transactions on Information Forensics and Security 15 (2020) 2750–2765.

[20] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, L. Cavallaro, Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed), in: Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, Asia CCS '19, ACM, New York, NY, USA, 2019, pp. 15–27.

[21] A. Afianian, S. Niksefat, B. Sadeghiyan, D. Baptiste, Malware dynamic analysis evasion techniques: A survey, ACM Comput. Surv. 52 (6) (Nov. 2019).

[22] A. Bulazel, B. Yener, A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web, in: Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium, ROOTS, Association for Computing Machinery, New York, NY, USA, 2017.

[23] J. Blackthorne, A. Bulazel, A. Fasano, P. Biernat, B. Yener, Avleak: Fingerprinting antivirus emulators through black-box testing, in: 10th USENIX Workshop on Offensive Technologies (WOOT 16), 2016.

[24] C. S. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation-tools for software protection, IEEE Transactions on software engineering 28 (8) (2002) 735–746.

[25] M. N. Gagnon, S. Taylor, A. K. Ghosh, Software protection through anti-debugging, IEEE Security & Privacy 5 (3) (2007) 82–84.

[26] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, J. Nazario, Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware, in: 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), 2008, pp. 177–186.

[27] R. R. Branco, G. N. Barbosa, P. Drimel, Scientific but not academical overview of malware anti-debugging , anti-disassembly and anti-vm technologies, in: Black Hat USA, Las Vegas, NV, 2012.

[28] G. N. Barbosa, R. R. Branco, Prevalent characteristics in modern malware, in: Black Hat USA, Las Vegas, NV, 2014.

[29] Y. Oyama, Trends of anti-analysis operations of malwares observed in api call logs, Journal of Computer Virology and Hacking Techniques 14 (1) (2018) 69–85.

[30] P. Chen, C. Huygens, L. Desmet, W. Joosen, Advanced or not? a comparative study of the use of anti-debugging and anti-vm techniques in generic and targeted malware, in: J.-H. Hoepman, S. Katzenbeisser (Eds.), ICT Systems Security and Privacy Protection, Springer International Publishing, Cham, 2016, pp. 323–336.

[31] CheckPoint, Evasions: CPU, https://evasions.checkpoint.com/techniques/cpu.html.

[32] Paranoid Fish (Pafish), https://github.com/a0rtega/pafish.

[33] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, G. Wang, Measuring and Modeling the Label Dynamics of Online Anti-Malware Engines, in: USENIX Security Symposium), 2020.

[34] M. Sebastián, R. Rivera, P. Kotzias, J. Caballero, Avclass: A tool for massive malware labeling, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2016, pp. 230–253.

[35] A. Küchler, A. Mantovani, Y. Han, L. Bilge, D. Balzarotti, Does Every Second Count? Time-based Evolution of Malware Behavior in Sandboxes, in: Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS), 2021.

[36] C. Rossow, C. J. Dietrich, C. Grier, C. Kreibich, V. Paxson, N. Pohlmann, H. Bos, M. Van Steen, Prudent practices for designing malware experiments: Status quo and outlook, in: 2012 IEEE Symposium on Security and Privacy, IEEE, 2012, pp. 65–79.

[37] O. Kulchytskyy, A. Kukoba, Anti debugging protection techniques with examples, https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software (2019).

[38] Microsoft, Determining if a debugger is attached, https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/determining-if-a-debugger-is-attached (mar).

[39] P. Ferrie, The ultimate anti-debug reference, https://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf (2011).

[40] Cyberbit, Anti-vm and anti-sandbox explained, https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/ (2007).

[41] M. Sikorski, A. Honig, Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, 1st Edition, No Starch Press, San Francisco, CA, USA, 2012.

[42] D. Instinct, Common anti-debugging techniques in the malware landscape, https://www.deepinstinct.com/2017/12/27/common-anti-debugging-techniques-in-the-malware-landscape/ (2017).

[43] Virtual machine detection techniques, https://shasaurabh.blogspot.com/2017/07/virtual-machine-detection-techniques.html (2017).

[44] McAfee, Evolution of malware sandbox evasion tactics – a retrospective study, https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/ (2019).

[45] Y. Oyama, Investigation of the diverse sleep behavior of malware, Journal of Information Processing 26 (2018) 461–476.

[46] C. Kolbitsch, E. Kirda, C. Kruegel, The power of procrastination: Detection and mitigation of execution-stalling malicious code, in: Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS '11, ACM, New York, NY, USA, 2011, pp. 285–296.

[47] FireEye, Turing test in reverse: New sandbox-evasion techniques seek human interaction, https://www.fireeye.com/blog/threat-research/2014/06/turing-test-in-reverse-new-sandbox-evasion-techniques-seek-human-interaction.html (2014).

[48] FireEye, Increased use of a delphi packer to evade malware classification, https://www.fireeye.com/blog/threat-research/2018/09/increased-use-of-delphi-packer-to-evade-malware-classification.html (2018).

[49] D. Instinct, Anti-virtualization malware, https://www.deepinstinct.com/2019/10/29/malware-evasion-techniques-part-2-anti-vm-blog/ (2019).

[50] U. Project, Anti-debugging, http://unprotect.tdgt.org/index.php/Anti-debugging (2017).

[51] FireEye, Increased use of wmi for environment detection and evasion, https://www.fireeye.com/blog/threat-research/2016/10/increased_use_ofwmi.html (2016).

[52] DeepInstinct, Anti-virtualization malware, https://www.deepinstinct.com/2019/10/29/malware-evasion-techniques-part-2-anti-vm-blog/ (2019).

[53] U. Project, Sandbox evasion, http://unprotect.tdgt.org/index.php/Unprotect_Project (2017).

[54] PEiD, https://www.aldeid.com/wiki/PEiD.

[55] T. Timzen, Kernel Forensics and Rootkits, https://www.tophertimzen.com/resources/cs407/slides/week06_01-Rootkits.html.

[56] McAfee, McAfee Labs Threats Report, https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf (June 2017).

[57] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, G. Vigna, Triggerscope: Towards detecting logic bombs in android applications, in: 2016 IEEE symposium on security and privacy (SP), IEEE, 2016, pp. 377–396.

Appendix A. Evasive Techniques: Details

Appendix A.1. Memory Fingerprinting

A debugger usually leaves some traces in the memory of the debugged process. We consider some memory locations that are interesting for Anti-Debug checks. Some of them are inside the Process Environment Block (`PEB`) or the `KUSER_SHARED`[3], which are Windows data structures inside every running process. Our *Memory Controller* module handles all the memory accesses. We consider the access evasive only if the load instruction is executed inside the main binary code section to filter relevant results. In particular, we consider the following memory accesses: `PEB→IsDebugged` is a byte inside the PEB, checked by `IsDebuggerPresent` [37, 26], which is 1 if the process is inside a local debugger and 0 otherwise; `PEB→NtGlobalFlag` is a byte inside the PEB at offset `0x68` and it is 0 when no debugger is in place and `0x70` otherwise [37]; `PEB→Heap→Flags` is a byte inside the Process Heap (offset `0x18` from PEB) and located at offset `0x40` from the Process Heap, which is always different from 2, when under a debugger [37]; `PEB→Heap→ForceFlags` is a byte inside the Process Heap (offset `0x18` from PEB) and located at offset `0x44` from the Process Heap, which is always different from zero when under a debugger [37]; `KUSER_SHARED→KdDebuggerEnabled` is a byte inside `KUSER_SHARED`, which is 1 when a kernel debugger is detected at boot time, 0 otherwise.

Appendix A.2. Error/Exception Handling

Malware samples can exploit exceptions and errors to identify the presence of a debugger. Indeed, a malware sample sets a custom handler, then throws an exception, and it executes its malicious activities inside the unhandled handler. There are several ways to implements such behavior [39]. Debuggers usually catch `CTRL+C` exception using their own handles (`CTRL` **Exception Handling**). Therefore, a debugger can be detected throwing a `CTRL+C` exception after having set a handler for the exception. To detect this behavior, our tool checks if a `SetControlCtrlHandler` is called before throwing this exception using `GenerateConsoleCtrlEvent`. Similarly, `SetUnhandledExceptionFilter` is used to set the last handler in the context of

[3] `KUSER_SHARED` is a memory region that contains data shared between kernel and userspace [38].

the Structured Exception Handling (SEH) mechanism in Windows. The *Unhandled Exception Handler* is used to perform the last recovery action before killing the process. Windows uses the standard Windows Frame as the last exception handler. However, debuggers ignore this exception to keep processes running even when an exception occurs. Moreover, a malware sample can exploit the close of an invalid Handle (`NtClose(INVALID_HANDLE)`) that throws the exception with code `(0xc0000008)`. Slightly different is the use of `OutputDebugString`. This technique does not work on modern Windows systems; however, it was used in the past. It consists of a call to `OutputDebugString`, which sets an error if a debug is not present. It is possible to recover information about the error using `GetLastError` function. In the past, `OutputDebugString` was used to crash debuggers by using a %s format string as an argument. Our tool detects any `OutputDebugString` call during the execution.

Appendix A.3. CPU Fingerprinting

`CPUID` is an x86 instruction used to retrieve information about the running CPU. It is commonly used to retrieve information about the CPU: supported extensions, CPU brand, Floating-Point unit and cache description, CPU serial number, and other information. This instruction is generally used by computation-intensive programs that need to know, for example, if SSE and floating-point operations are supported natively. There are two common ways to detect a hypervisor using this instruction [40]: `CPUID(EAX=0x00000001)` returns information about the presence of a Hypervisor. Instead, `CPUID(EAX=0x40000000)` returns the brand of the hypervisor, such as *WMware* or *VBOXVBOX*.

`IN` is a privileged x86 instruction used to communicate with peripherals and takes as arguments EAX and an immediate or EAX and EDX. Inside some hypervisors, the instruction can be executed in user-mode with some specific port values (EDX) to communicate with the hypervisor, which sets some special I/O ports available to guests. VMware supports this feature, indeed, `IN(EAX='VMXh', EDX='VX')` results into `EBX='VMXh'` if executed under a VMware Hypervisor, thus this method actively exploited by evasive malware to determine if they are under a VM [40].

`SMSW` has an undocumented bits for the returned value. Some VMware versions have a default value for these undocumented bits [41].

Appendix A.4. Table Descriptors

These techniques find discrepancies between OS descriptor tables addresses in order to detect the hypervisor presence.
`STR` returns the pointer to the Task State Segment(TSS) of the current task. Its value is fixed on single-core physical machines but changes in virtual machines. Moreover, `SIDT` is an instruction used to retrieve the Interrupt Descriptor Table (IDT). Old VMware did not emulate this instruction correctly, so the value of the Hypervisor IDT could be leaked. In particular, if a single CPU machine is in place and a VMware Hypervisor is present, it is possible to notice two different IDT addresses for the same CPU. These methods do not work reliably on multi-CPU systems [41].
`SGDT` is used to retrieve the Global Descriptor Table (GDT), which should be unique; however, while running on VMware Hypervisor, the value keeps changing. The difference can be detected querying multiple times `SGDT` [41]. Instead, `SLDT` is used to retrieve the Local Descriptor Table (LDT). This value is always zero on physical Windows machines because the Windows OS never uses it. Although, while running on VMware, LDT value changes [41].

Appendix A.5. Traps

Several x86 instructions can be used to leak the presence of a hypervisor, emulator, or debugger [39]. Similarly, to Exception Handling (see Section Appendix A.2) we can exploit `INT 3` that throws a `BREAKPOINT_EXCEPTION` that is used by debuggers to set breakpoints inside a program, or `INT 1` that is used for is a `SINGLE_STEP` exception [30]. `ICEBP` (opcode 0xf1) is an undocumented x86 instruction that trigger `SINGLE_STEP` exception [41]. Another way to trigger exception is to set the trap flag inside `EEFLAGS` register. In fact, it possible to pop a new value for `EEFLAGS` using `POPF` and `POPFD` instructions [37]. Other instructions like `POP SS`, instead, tampers with the step execution of debuggers skipping the next instruction [37]. `INT 2D` throws a `BREAKPOINT_EXCEPTION`, and in some debugger (OllyDBG), this instruction will skip the one byte causing a different instruction to be executed.
`VPCEXT`, instead, is an instruction that can be executed only by Virtual PC Emulator, and it throws an exception if executed outside a Virtual PC. Some malware samples execute `VPCEXT` and then check the value of EBX to understand if Virtual PC detection succeeded [43]. Virtual PC is right now

an abandoned project[4](last version released in 2011). Thus this technique is unreliable on modern systems. For all these techniques, our system detects when corresponding instruction is executed.

Appendix A.6. Timing

It is possible to exploit accurate timestamps to identify analysis systems. For example, discrepancies in the execution time of instructions are clear signs of a virtual environment. However, these techniques can also be used to detect debuggers and DBI tools [18]. Indeed, any operation that slows down the execution of the sample can be detected.
`RDTSC` is an x86 instruction used to retrieve the number of clock cycles since reset. It is a 64-bit value used for performance measurements. In order to identify the presence of a Hypervisor, it is sufficient to execute VM Exit instructions such as `CPUID` while measuring the timing with `RDTSC`. We can detect the Virtual Machine by looking at the number of cycles needed to execute such instructions. `RDTSC` can also be used to measure the time elapsed between two blocks of code for debuggers or DBI detection [41].

We implemented a system to bypass our VM detection by saving the first `RDTSC` return value. All the subsequent calls are set to a small increment with respect to the previously saved value.

There is a series of Windows API calls that can be used for timing detection: `QueryPerformanceCounter` and `NtQueryPerformanceCounter` are unbiased versions of `RDTSC` that take into account multiprocessors and frequency changes across CPUs; `timeGetTime`, `timeGetSystemTime`, and `GetSystemTimes` return the time since system boot; `NtQuerySystemTime` and `GetLocalTime` return the current date in different formats; `GetTickCount`, `GetTickCount64`, and `NtGetTickCount` return the number of milliseconds since boot; `QueryInterruptTime`, `QueryInterruptTimePrecise`, `QueryUnbiasedInterruptTime`, and `QueryUnbiasedInterruptTimePrecise` return the number of 100 nanoseconds units since system boot using clock interrupts. All these functions are hooked and logged by our detection system. However, to bypass the detection of an analysis environment, all the internal Windows timers should be set properly.

`GetTickCount` and `timeGetTime` can be used to detect how much time the system has been up, and some malware sample uses them to detect the

[4] https://www.microsoft.com/it-it/download/details.aspx?id=3702

uptime of the whole system. Indeed, a small uptime is considered an indication of Sandbox [44]. Initially, our detection system sets timing variables to a value compatible with 7.7 hours of uptime and increases their value by a small constant for any subsequent timing information retrieval.

Timing evasion techniques usually involve the repeated measurement of performance counters and ticks to check if a significant amount of time is spent on some instructions. If the delta between two measurements is less than a threshold, the system is deemed clean. Setting always a small difference between measurements bypasses all the possible timing checks.

In order to avoid the manual implementation for all the timing functions, we applied a bypass based on monitoring access to all the timing fields contained in `KUSER_SHARED`: `TickCountQuad` contains the value of ms elapsed since the system boot. This value is used to to produce `GetTickCount`, `GetTickCount64` , and `NtGetTickCount`. The initial value is set to $\approxeq$ 28000000 ticks ($\approxeq$ 7.7 hours); `InterruptTime` is the field accessed by `timeGetTime` and `QueryInterruptTime` and it is a more precise way to get the ticks of the system. This field is initialized with the value of `TickCountQuad` and incremented with a higher frequency: `SystemTime` is the field containing the value of the system date with millisecond precision. Each time any of the previous timing fields are accessed, our tool adds a tiny constant to increase the timer. Using this solution, a malware sample that continuously accesses timing information cannot reliably tell the difference between physical and virtual environments.

Appendix A.7. Stalling

Malware Sandboxes are designed to detonate malware inside the Virtual Environment in order to collect system events. However, this activity is limited by the time assigned to the analysis. A reasonable time assigned by Sandboxes ranges from 3 minutes to 10 minutes,; thus, a safe choice is to start malicious actions after 10 minutes in order to fool the detection system. This behavior is extremely stealthy. Understanding if a sample is stalling is an undecidable problem. A malware sample does not stall only using sleeping functions offered by the OS, but also performing useless arithmetic operations, or system calls [46, 45]. Even worse is the case of malware that contains time bombs that run only at specific dates or timestamps. Our system is limited, focusing only on the functions provided by Windows. The main sleeping function is `Sleep`, which stalls the execution of a program. `Sleep` internally

calls `SleepEx` which in turn executes the syscall `NtDelayExecution`. Moreover, we can detect the use of: `waitForSingleObject/Ex`, used to wait until an event is signaled. However, it is possible to set a maximum waiting time, which causes an infinite stalling if the signal is never sent; `SetTimer`, `timeSetEvent`, `SetWaitableTimer`, used to set the execution of a function after the timeframe is elapsed; `CreateTimerQueueTimer` creates a timer inside a Queue of timers. When the timer expires a callback function is executed; `IcmpSendEcho/2/Ex` is used to wait for the response of ICMP echo responses with a timeout. If the server never replies, the function will stall until the timer expires.

We set to 0 the sleeping time defined as a parameter of the aforementioned functions. This countermeasure is applied if the function is invoked by the text segment of the malware executable because most of the sleeping functions such as `waitForSingleObject` and `SetTimer` are used for synchronization purposes by external libraries whose functionalities could be broken with a null value.

Appendix A.8. Human Interaction

The idea of this set of techniques is to detect if a human is using the infected machine. A Virtual Environment can be detected checking the mouse cursor position over time using `GetCursorPos` API call (**Cursor Position**). If the position of the cursor never changes, a malware sample can reliably say that it is running inside a Sandbox [47]. Our tool detects this API call and randomizes the coordinates to bypass the evasive behavior.

Another way to detect the presence of the user on the system is to retrieve the timestamp of the last input. This method has been detected on custom Delphi packers [48]. The last input is measured across multiple Human Interface Devices (HID) such as mouses and keyboard. If the timestamp is old, the malware can say that no user is present on the system. The common Windows API call to get this information is `GetLastInputInfo` which returns the timestamp of the last input action in system ticks. Each tick is roughly 1 ms and the timestamp is zero when the system boots. Our tool logs the call and sets the returned value of `GetLastInputinfo` to the following value: $LastInput = CurrentTicks() + rand()\%1000$

Appendix A.9. Registry

The Windows Registry is a container of valuable system information. It contains interesting values such as Services available, Programs Installed and

System information that can also be retrieved using Windows API calls. The Registry is a hierarchical database where each key stores some values. Each value has some typed data associated that can be queried using Windows Registry Editor or programmatically using Windows Libraries. The type of detection is similar to the one described for File System Artifacts [49]. Indeed, we use this blacklist to log and then block suspicious Registry Keys. In particular, we focus on the most reliable techniques to retrieve Registry information.

`NtOpenKey/Ex` The only two syscalls available to open a registry key. The existence of a VM/Debugger related key is a good indicator of a malware analysis system. There are some VirtualBox specific Keys that are never present on physical machines. Our tool allows the generation of a valid `HANDLE` if the full key path does not contain blacklisted keywords, otherwise `STATUS_OBJECT_NAME_NOT_FOUND` is returned. An example of blacklisted key is: `HKLM\HARDWARE\ACPI\DSDT\VBOX__`.

`NtQueryValueKey` It takes the HANDLE of an open key and a value name and returns the data stored in the registry. If the value name contains some blacklisted keywords, the result is a non-existing value error. If the data stored contains a blacklisted keyword, the whole data is replaced with a random string. This implementation takes into account String (`REG_SZ`, `REG_EXPAND_SZ`) and MultiStrings (`REG_MULTI_SZ`) values.

`NtEnumerateKey` It enumerates all the subkeys of a key. Again, if some subkey should be blacklisted, its name is replaced with random values to hide its presence.

`NtEnumerateValueKey` It is used to enumerate all the values and data contained in a key. In this case, the same countermeasure of `NtEnumerateKey` is applied, taking into account `REG_SZ`, `REG_EXPAND_SZ`, `REG_MULTI_SZ` values.

Appendix A.10. System Environment

Physical Memory Information. Malware can detect analysis VMs by checking memory size information, because they are usually configured to take few GB of memory. An evasive behavior consists in checking if the physical memory size is significantly less than 4 GB [53]. There are two ways to obtain this information: First, using `NtQuerySystemInformation(PHYSICAL_MEMORY_INFO)`, an undocumented syscall. Second, using `GlobalMemoryStatusEx`,a Windows API call that returns a documented and detailed report on the memory status, including free memory and total memory size.

Disk Space Information. The size of the disk installed in the system is a good indicator of VMs, because most VMs do not take more than 80 GB [53]. There are three documented ways to obtain information about the disk size. `GetDiskFreeSpaceEx/A/W` returns the total space on disk and free bytes; `GetDiskFreeSpace/A/W` returns the same information and additional specifics on Sectors and Clusters sizes; `DeviceIoControl(IOCTL_DISK_GET_LENGTH_INFO)` is used to send control codes to device drivers. We use `IOCTL_DISK_GET_LENGTH_INFO` to ask for Disk Information.

The detection is done using the Windows API Hooker module, and the countermeasures applies a Disk Space Multiplier on the total size of the disk. We set this multiplier to 5, e.g., if the analysis VM has 30 GB, the API call will return 150 GB.

Network Information. It is widely known by malware developers that MAC Addresses have specific starting bytes depending on the Hypervisor Brand and the same Virtual NIC names are used across multiple VMs configurations. For example VirtualBox uses these starting bytes `08:00:27` for every NIC installed on the Guest OS [40]. An evasive malware simply can retrieve the MAC address of any NIC installed and perform a comparison on the first 3 bytes. NIC information can be retrieved by either calling `GetAdaptersInfo` or `GetAdaptersAddresses`. The two aforementioned Windows API calls are Hooked, Logged and the returned MAC Addresses and NIC Names are set to values defined by the analyst. For our analysis we used a MAC address corresponding to Intel Corporate Adapters and a NIC name that resembles the name of a real network card.

Computer Name detection. The computer name information is used to fingerprint sandboxes. To evade specific sandboxes, malware developers build a list of sandboxes computer names and then they use the functions `GetComputerName/Ex/A/W` to retrieve this information. Our tool intercepts attempts to retrieve this information.

`IsDebuggerPresent.` A Windows API that returns True if a debugger is attached to the process [41, 14].

`CheckRemoteDebuggerPresent.` This checks if the process is under a debugger in separate and parallel process [41, 14].

`NtQuerySystemInformation(0x23).` Syscall that returns a structure containing a flag set to true when a Kernel Debugger is in place [41].

`NtQueryObject.` This technique makes use of undocumented system calls. The program begins creating a debug object using `NtCreateDebugObject`,

and then the handle to that object is queried to see how many debug objects are present. In principle, there should be a debug object for each debugged process, however if a debug object is created and a debugger is present, two debug objects should be present. This is a very powerful feature and can be exploited also for anti-anti-debug checks. Our detection is to log the action if a `NtQueryObject` syscall is executed after a `NtCreateDebugObject`. [50]

`FindWindow.` This technique is commonly used to detect Debuggers and Analyzers, and returns an Handle to the window corresponding to the window name specified as a parameter. Using this technique it is possible to know if specific tools are running on the system [41].

Number of Processors. Nowadays most machines have more than 1 CPU, however it is not uncommon to find VM configurations with a single CPU. Thus, the number of processors has been exploited as a feature by evasive malware to detect VM presence [44]. This information can be retrieved using Windows API calls, but the way most API calls get this value is reading a field inside the Process Environment Block (PEB) inside the Windows process address space. The number of CPU is at the `PEB_BASE_ADDRESS + 0x64`. Our tool using the Memory Accesses Controller monitors any access to this memory location logging the access and setting the value to 4. This way, any malware will detect a 4 CPU machine.

Appendix A.11. WMI

Windows Management Instrumentation is a proprietary technology by Microsoft for Enterprise Management of Windows Machine. It is possible to write scripts for machines maintenance and retrieve their configurations. This last aspect can be exploited for evasion purposes. Indeed, `WMI` framework allows C++ developers and system administrators to write `WMI` Queries in `WQL` which is a `SQL-like` language for `WMI` information. These queries will return any kind of System related information. For example, performing `SELECT * FROM Win32_BIOS` it is possible to retrieve information about the BIOS Manufacturer which is specific for each Hypervisor. Other information can be extracted for fingerprinting purposes such as: process list, system resources, AVs installed and more [51]. The main method to execute a query is `ExecQuery` which requires the query string in the `WQL` language. Our bypass methodology is based on denying and logging every `WMI` query attempt. Indeed, it is unfeasible to understand from `WQL` queries which specific information the malware is searching.

Appendix A.12. Process Environment

`NtQueryInformationProcess` is a system call to retrieve information about the running process. Three Process Information Classes (under the form of integers) can be exploited to detect a debugger. `NtQueryInformationProcess(0x07)` is the underlying system call corresponding to the `CheckRemoteDebuggerPresent` API call [41]. `NtQueryInformationProcess(0x1e)` is undocumented but leaks the presence of a debugger with the same behaviour of `(1)` [37]. `NtQueryInformationProcess(0x1f)` is undocumented but behaves as `(1)` and `(2)` [37].

`NtSetInformationThread(0x11).` This is a system call used to prevent the thread from sending debugging events to the attached debugger [39].

`NtCreateThread(HIDE_FROM_DEBUGGER).` This is a system call used to tamper with debugging creating a thread that cannot be followed by the debugger [37].

`NtGetContextThread(CONTEXT_DEBUG_REGISTERS).` This is used to get the value of Hardware Debug Registers. When Hardware breakpoints are used, the returned values are always different from zero because breakpoint addresses are set [37].

`NtSetContextThread(CONTEXT_DEBUG_REGISTERS).` This is used to set the value of Hardware Debug Registers. In order to tamper with debugging it is possible to set to 0 every Debug Register in order to miss breakpoints [37].

`DebugActiveProcess` **on Parent Process.** A process cannot be debugged by two debuggers, so if a debugger is in place a child process with debugging privilege should fail its attempt to debug the parent process. Using this principle, the running process creates a child process and then the child process calls `DebugActiveProcess` on the parent. Therefore, a failed attempt to debug the parent is a clear indicator of the debugger presence [39]. To handle this case our tool supports forking. The monitoring activity is extended to the child process and when `DebugActiveProcess` is called on the parent process, the action is logged through the Windows API Hooker.

Appendix A.13. File System

An evasive behavior is to check inside known directories if a resource is present or not. For example, in Windows VirtualBox components are usually located under `C:\Program Files\Oracle\VirtualBox Guest Additions\`. A successful attempt to open this directory is an indicator of VM presence [49]. This same concept is applied to devices and files. There are

multiple ways to access File System resources: First, using `NtOpenFile`, a syscall to open files, directories or devices. On success it returns an handle. When a blacklisted resource is opened, our tool makes the open fail with the return value `STATUS_OBJECT_NAME_NOT_FOUND`. Second, `NtCreateFile` is a syscall designed to supersede `NtOpenFile` and it has the same functionalities, but it can also create new files and directories. The same behavior as `NtOpenFile` is applied when a blacklisted resource is created or opened. Third, `NtQueryAttributesFile` returns information about creation time, last access and write time, file size and file name length. Depending on the required attributes, file information can be more detailed. This syscall fails when the file is non existent, so a stealthier approach is to call it to check file presence. Also in this case our tool returns `STATUS_OBJECT_NAME_NOT_FOUND` when a blacklisted resource is detected. Finally, `NtQueryDirectoryFileEx` is a system call used to list the files contained in a directory. Before returning the whole list of files, our tool checks if some of them contain blacklisted keywords. In the latter case, the resource name is replaced with a random string.

Appendix A.14. List Processes

An easy way to detect debuggers, VMs and generic analysis tools is the process enumeration method [49]. The list of the processes can be retrieved using two methods. First, by calling `NtQuerySystemInformation(SYSTEM_PROCESS_INFORMATION)`, where `SYSTEM_PROCESS_INFORMATION = 5`, which is usually called by some high-level Windows API functions such as `CreateToolhelp32Snapshot`. Second, using the `EnumProcesses` Windows API, followed by `GetModuleBaseName`. Our tool is able to log attempts to retrieve the process list substituting at runtime the process names in blacklist with random strings, thus, fooling malware.

Appendix A.15. List Services

Services Enumeration. When a Guest OS is installed, hypervisor-related artifacts (e.g., VirtualBox Guest Additions) could be needed to work properly. These components could run as services inside the OS. Malware enumerates services to search for possible VM components [49]. This behavior works as follows. The Service Manager is opened via `OpenSCManager` Windows API call with the service enumeration permission. The actual Service Enumeration is performed using `EnumServicesStatus`, which returns a list of Service Names in the form of Keys for the Services Manager Database

and a human readable format. If a malware sample knows exactly what it is searching for it can call `GetServiceKeyName`, `GetServiceDisplayName` or `OpenService` after `OpenSCManager`. The first requires the human readable name of the service, the second one requires the name assigned by the Service Manager Database, while the last one is used to open an existing service based on its name. We log the aforementioned calls and, if `EnumServicesStatus` reveals a name contained in our blacklist, the service name is substituted with random Display and Key names, while when `OpenService` is called on a blacklisted service, the action fails. If the detection of running VM services is done using `GetServiceKeyName` or `GetServiceDisplayName`, the value returned is an empty string, as specified by the documentation in case of failure.

Appendix A.16. Drivers Information

Drivers Enumeration. The Drivers enumeration evasion technique follows the same principle of Services Enumeration. In order to work correctly most Guest OS contains drivers related to Virtual Peripherals built by the Hypervisor. These components can be enumerated either by direct access to the Windows Driver folder or using a Windows API call. The first method can by handled blocking access to VM files to the malware as we will discuss in Subsection Appendix A.13. Some examples of drivers checked by malware is described in [52]. The second method exploits `EnumDeviceDriver` API call. An initial call is done to `EnumDeviceDriver`, which returns an array of drivers addresses. For each of them a call is done to `GetDeviceDriverBaseName` in order to get the Driver name. The Anti-Evasion System intercepts `EnumDeviceDrivers` as an action and replace any VM-related driver name returned by `GetDeviceDriverBaseName` with a random string.