Malware Dynamic Analysis Evasion Techniques: A Survey

AMIR AFIANIAN, SALMAN NIKSEFAT, and BABAK SADEGHIYAN, APA Research Center, Amirkabir University of Technology, Iran
DAVID BAPTISTE, ESIEA (C + V)O Lab, France

The cyber world is plagued with ever-evolving malware that readily infiltrate all defense mechanisms, operate viciously unbeknownst to the user, and surreptitiously exfiltrate sensitive data. Understanding the inner workings of such malware provides a leverage to effectively combat them. This understanding is pursued often through dynamic analysis which is conducted manually or automatically. Malware authors accordingly, have devised and advanced evasion techniques to thwart or evade these analyses. In this article, we present a comprehensive survey on malware dynamic analysis evasion techniques. In addition, we propose a detailed classification of these techniques and further demonstrate how their efficacy holds against different types of detection and analysis approaches.

Our observations attest that evasive behavior is mostly concerned with detecting and evading sandboxes. The primary tactic of such malware we argue is fingerprinting followed by new trends for reverse Turing test tactic which aims at detecting human interaction. Furthermore, we will posit that the current defensive strategies, beginning with reactive methods to endeavors for more transparent analysis systems, are readily foiled by zero-day fingerprinting techniques or other evasion tactics such as stalling. Accordingly, we would recommend the pursuit of more generic defensive strategies with an emphasis on path exploration techniques that has the potential to thwart all the evasive tactics.

CCS Concepts: • **Security and privacy** → *Malware and its mitigation*; *Systems security*; *Operating systems security*;

Additional Key Words and Phrases: Malware, evasion techniques, anti-debugging, sandbox evasion

ACM Reference format:
Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. 2019. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Comput. Surv.* 52, 6, Article 126 (November 2019), 28 pages.
https://doi.org/10.1145/3365001

1 INTRODUCTION

Gaining access through ARPANET network, Creeper copied itself remotely to other computers, and, prompted the message: "Catch me if you can." Emerged as an experiment in 1971 [23], *Creeper* carried no harmful purpose, yet, with its quick propagation it cast the light on a future with individuals or entities incorporating their malicious intentions into a software (hence the name malware),

This work is supported by APA research center (http://apa.aut.ac.ir) at Amirkabir University of Technology, Tehran, Iran.
Authors' addresses: A. Afianian, S. Niksefat, and B. Sadeghiyan, APA Research Center, Amirkabir University of Technology, No. 350, Hafez Ave, Valiasr Square, Tehran, Iran 1591634311; emails: {a.afianian, niksefat, basadegh}@aut.ac.ir; D. Baptiste, ESIEA (C + V)O lab, Laval, France; email: baptiste.david@esiea.fr.

0360-0300/2019/11-ART126 $15.00
https://doi.org/10.1145/3365001

that are capable of inducing undesired and harmful effects on the infected systems [11]. In those days, developing malware was merely about showing off; but, in today's world, as the Internet has integrated every individual, institute, and organization into a single cohesive complex, the goal of malware authors has extended to include far more lucrative objectives, i.e., money, intelligence, and power. Accordingly, these possibilities have profoundly motivated a new malware industry, the outcome of which, is highly sophisticated variants, which aptly infiltrate systems and operate viciously unbeknownst to the user or defense mechanisms.

Devising countermeasure or technologies that can withstand this level of sophistication would be possible merely by understanding the precise inner workings of such malware. Malware analysis is the way to achieve this understanding [32]. Initially, with the help of disassemblers, decompilers, and so forth, analysts inspected the malware's binary and code to infer its functionality. This approach, which is also referred to as static analysis, became far more arduous and intricate with the advent, and evolution of code obfuscation tactics [16, 111, 117, 146] and other evasion tactics targeting static analysis (e.g., opaque constants [96], packers [50]). As a resolution, a promising approach that was adopted was dynamic analysis in which the basis of analysis and detection is what the file does (behavior) rather than what the file is (binary and signature) [97]. Put differently, in dynamic analysis, an instance of the suspected program is run and its behavior is inspected in runtime. This approach would obviate the hurdles posed by the aforementioned static analysis evasion tactics. To thwart these efforts, however, malware authors turned to a new category of evasion tactics that targeted dynamic analysis.

In this article, we identify two modes of dynamic analysis, i.e., manual and automated. The manual dynamic analysis is comprised of the traditional form of dynamic analysis and is often conducted with the help of debuggers. And more recently, manual dynamic analysis is aided by more transparent and bare-metal systems. The automated dynamic analysis is a more novel approach and also a response to the ever-increasing new samples that security vendors face on a daily basis. The automated dynamic analysis is often represented by the Sandbox technology. Further, we conduct a comprehensive survey on evasion techniques that are tailored to each mode of analysis. Additionally, we will have a brief survey on countermeasure tactics against evasive malware that the defensive industry is pursuing.

Moreover, we will portray the current and also emerging evasive trends. In the case of manual dynamic analysis evasion, we posit that anti-debugging techniques are still significantly being practiced. Also, we will discuss how the new fileless malware is emerging and prevailing over the traditional dynamic analysis. Furthermore, we divide all the evasion tactics (tailored to both manual and automated) into two broad categories: detection-dependent, and detection-independent. Following a detection-dependent strategy for evasion, malware incorporates techniques to explicitly detect its execution environment. A more different strategy is detection-independent in which the malware behaves the same, regardless of its execution environment. In other words, to evade analysis, malware does not have to detect the execution environment.

Also, We argue that until recent times, sandbox technology has been focusing on technologies that chiefly target the detection-dependent evasion tactics. As a response, malware authors are gradually adopting more and more detection-independent tactics to evade such automated dynamic analysis environments.

Our goal in this article is threefold. First, given the importance of the subject, we have aimed at offering a comprehensive classification, with the hope of moving toward an established taxonomy. Second, we aspired to unveil the direction toward which the evasive behavior is trending. Thus, we tried to yield a view of the current situation for the pervasiveness of each evasion tactic. Our third goal is to identify the shortcomings of the current defensive strategies and offer a direction which we deem to have more potential to effectively confront evasive malware.

Our contributions are as follows:

—We present a comprehensive survey of malware dynamic analysis evasion techniques for both modes of manual and automated. In [21], authors have conducted a comprehensive analysis on dynamic analysis evasion on different platforms. However, their focus is on automated dynamic analysis and the discussion of manual dynamic analysis evasion is rather brief. Other surveys like [43, 81] trivially surveyed analysis evasion and provide no detailed overview of malware dynamic analysis evasion.
—For both manual and automated modes, we present a detailed classification of malware evasion tactics and techniques. To the best of our knowledge, this would be the first comprehensive survey of dynamic analysis evasion tactics that offers a thorough classification.
—We portray the current trends in the realm of automated dynamic analysis evasion techniques and discuss the proper course of action for future directions.
—We provide a brief survey on countermeasures against evasive malware that the industry and academia is pursuing. In addition, we yield a classification for such endeavors.

1.1 Overview

1.1.1 Scope. Latest reports attest to the dominance of Windows malware by a staggering ratio of 77.22% in 2017 [5]. Statistics also show that due to the employment of evasion techniques, "64% of AV scanners fail to identify 1% of the malware after 1 year [67]." These figures have further motivated us to converge our focus on evasive malware targeting Windows OS. Moreover, in our research, we have striven to probe the intersection of academia and industry. Thus, we have reviewed both academic papers, and related industry-provided literature.

1.1.2 Approach. Our primary endeavor has been to provide a detailed and vivid overview of dynamic analysis evasion techniques. To this end, we have classified the results of our scrutiny in three levels: category, tactic, and technique. Each category represents a particular approach and a goal which is both shared and pursued through different tactics. Each tactic then is implemented via different techniques. Throughout the survey, we elaborate the details down to the tactic level and we discuss several representative techniques. At the end of each section, we provide a summarizing table.

1.1.3 Evaluation. To fulfill the goals of this survey, we have employed several criteria such as efficacy and pervasiveness of different techniques which serve as the basis of comparisons for the related topic. In our summarizing tables, we present data from reviewed works where statistics are available, and otherwise, we offer evaluations based on our own observation.

1.1.4 Organization. In Section 2, we discuss manual dynamic analysis (debugging) and corresponding anti-debugging tactics. In Section 3, we discuss the automated dynamic analysis and the corresponding evasion tactics. Sections 2 and 3 are followed by a discussion. Finally, Section 4 concludes the article. Figure 1 demonstrates our proposed classification of the malware dynamic analysis evasion tactics.

2 PRELIMINARIES

In this section, we define several keywords that we extensively use throughout the article. Some terms, specifically, we deemed to need explanation since the passage of time has overloaded or altered their pervasive meaning.

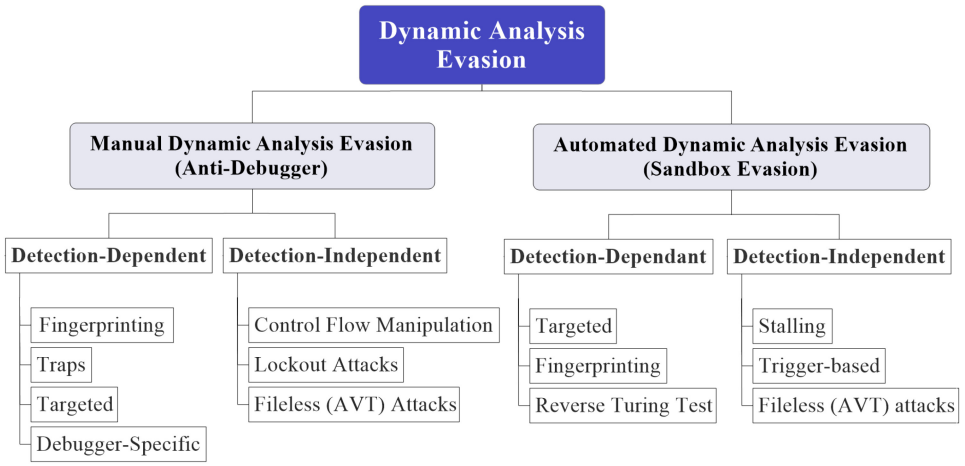

Fig. 1. A classification of malware dynamic analysis evasion tactics.

2.1 Sandbox

Traditionally, sandbox was developed to contain unintended effects of an unknown software [46]. Hence, the term sandbox meant an isolated or highly controlled environment used to test unverified programs. Due to similarities in nature, virtualized machines and emulated environments are often seen to be called sandbox. But, in this article, we use the term sandbox to refer to the contained and isolated environments that analyze the given program *automatically*, without the involvement of a human. The dividing line for us, in this article, given the trend of the industry, is the autonomous nature of the system. For instance, for a modified virtualized machine such as *Ether*, we consider the term debugger rather than sandbox.

2.2 Evasion and Transparency

In literature, evasion constitutes a series of techniques employed by malware in order to remain stealth, avoid detection, or hinder efforts for analysis. For instance, a major evasion tactic as we will discuss is fingerprinting [103]. With fingerprinting, the malware tries to detect its environment and verify if it is residing in a production system or an analysis system. In the same level, one major strategy to counter the evasions is to hide the cues and clues that might expose the analysis system. A system is more transparent if it exposes fewer clues to malware [66].

2.3 Manual vs. Automated

Manual and automated analysis are two major terms which form the basis of our classifications. The manual analysis is when the analysis process is performed by an expert human with the help of a debugger. If a technique uses the traditional sandboxing technology to implement a transparent debugger, we consider them under the manual dynamic analysis evasion. Automated, on the other hand, is the process that is the similar analysis that is performed automatically by a machine or software, also known as sandbox.

2.4 Detection vs. Analysis

Previously, there was no need for defining these two terms. Detection would simply refer to the process of discerning whether a given file is malicious or not; analysis, on the other hand, would refer to the process of understanding how the given malware works. Today, however, this dividing

line is blurry. The reason is that the role of automated analysis tools such as sandboxes is now extended. In addition to reporting on malware behavior, sandboxes are now playing their role as the core of automated detection mechanisms [66]. In this article, we follow similar concepts when uttering *analysis*. For *manual* it would mean understanding the malware behavior [121]; for the automated, additionally, it can mean detection.

2.5 Static vs. Dynamic

There are two major types of analysis, i.e., static and dynamic. Static analysis is the process of analyzing the code or binary without executing it. Dynamic analysis is the process of studying the behavior of the malware (API, system calls, etc.) at the runtime. Both types of analysis can be performed either manually or automatically. In this article, our focus is on dynamic analysis and how malware tries to prevent or evade such analysis.

2.6 Category, Tactic, and Technique

Throughout this article, we use the terms *category*, *tactic*, and *technique* that are the basis of our classification. The category of evasion is our high-level classification. Each category has the goal of evasion with a specific attitude for achieving it. This attitude is highly correlated with the efficacy of the evasion and is pursued by the tactics under each category. Tactics, in other words, are the specific maneuvers or approach for evasion with the specified attitude of its parent category. Finally, techniques are the various practical ways of implementing those tactics.

3 MANUAL DYNAMIC ANALYSIS EVASION

As cited in the Introduction, due to the employment of code obfuscation, packers, and so forth, static analysis of malware has become a daunting task. To obviate the conundrums and limitations of this approach, analysts opt for dynamic analysis in which malware's behavior is inspected at the runtime and often with the help of debuggers. This way of examination has two major benefits; one is that it relieves us from the impediments inflicted by packers, polymorphism, and so forth Second, taking this approach enables us to explore the activities that manifest themselves only in runtime [110] such as the interaction of the program with the OS [97]. We view this approach which is aided by debuggers, under the term "manual dynamic analysis." The corresponding evasion tactics to this approach involve the set of employed techniques within malware code with the goal of hindering, impeding, or evading the analysis process. These measures include approaches such as detecting the presence of analysis tools on the system (e.g., Wireshark, TCPDump) or detecting virtual machines as a sign of analysis environment, but, the majority of manual analysis evasion techniques are targeted toward debuggers, which are the primary tools of manual dynamic analysis. Chen et al. ran a study on 6,222 malware samples to assess changes in malware behavior in the presence of virtualization or an environment with debuggers attached. They discovered in the presence of a debugger (not in a virtualized environment) around 40% of malware samples exhibited less malicious behavior [24]. The same study revealed that when malware are executed in a virtualized environment, merely 2% of the samples exhibit malicious behavior. This shows that even though there are similar tactics to evade both the automated and manual dynamic analysis, e.g., fingerprinting, the techniques are different. And sheer detection of virtualization or emulated environment does not suffice to evade manual analysis since more and more production systems are running on virtualized machines. Our coverage in this section includes the traditional (and still relevant) anti-debugging techniques to the more advanced, recent AI-powered techniques.

Simply, anti-debugging is the application of one or more techniques, to fend off, impede, or evade the process of manual dynamic analysis (debugging) or reverse-engineering. It is noteworthy to mention that anti-debugging techniques, similar to obfuscation, initially, were legitimate

practices conducted by developers to protect their software. This implies that anti-debugging has a long history which is followed by a wide and diverse set of techniques. In this section, we will present our survey on the most cited manual dynamic analysis evasion (anti-debugging) tactics and corresponding techniques [3, 17, 24, 36, 40, 97, 110, 121, 131, 143, 148] that are more relevant to the context of malware. In addition, for each technique, we employ four criteria to better yield an overview of their relative weight and efficacy. The four criteria for our comparison and our touchstone for evaluating them are as follows:

- **Complexity:** The difficulty of incorporating and implementing the technique within malware code. We associate low complexity if the technique involves simple API calls for detection of debuggers' presence, medium to those that require several calls to inquire information from the system as well and the need to deduce from the returned information the probability of debuggers presence, and high to those that are challenging to implement and require more than 30 lines of assembly code.
- **Resistance:** Resistance pertains to the difficulty level of counteracting the evasion technique. If it would be as simple as altering some system information (e.g., randomizing registry key or a bit in *PEB*) we consider it low; if defeating the anti-debugging technique requires user-level hook and *dll* injection, we assign medium resistance; if kernel-level manipulations, e.g., drivers, are needed we set the resistance level as high; and if the countermeasure requires operations below OS (virtualization, emulation, etc.) we consider it very high. Finally, we have N/A, associated with techniques for which there exist no countermeasures yet.
- **Pervasiveness:** Due to the extreme diversity of the malware sample dataset it is hard to establish a ground truth for the pervasiveness of each tactic. However, we strive to provide a fair view of this metric based on other works [17, 119] in addition to our own observations and experience in the field.
- **Efficacy-Level:** Implementation of a debugger can take different architectural paths, e.g., user-level, kernel-level, virtualization-based, and more recently, bare-metal. Under the efficacy-level criterion, we will note against which architecture of debugger the evasion tactic is effective. For the sake of brevity, we appoint 1 to refer to user-level debuggers, 2 to refer to kernel-level debuggers, 3 to refer to virtualization-based/emulation-based debuggers, and 4 to refer to bare-metal debuggers.

Additionally, we allot a column to *Countermeasure* under which we briefly note how the anti-debugging technique could be circumvented. We summarize our manual dynamic analysis evasion study in Table 1. Before scrutinizing the anti-debugging techniques, we will have a brief introduction on debuggers, their goals, types, and techniques. Building upon this understanding, we can better understand and elaborate anti-debugging tactics and what makes them feasible.

3.1 A Briefing on Debuggers

As Sikorski states, "A debugger is a piece of software or hardware used to test or examine the execution of another program [121]." Among the core functionality of debuggers [1], there are several features that are critical to malware analysis such as stepping through the code one instruction at a time, pausing or halting it on desired points, examining the variables, and so forth. To provide each of the mentioned functionalities, debuggers rely on different tactics and each tactic is often aided with specific hardware or software provisions that inevitably result in subtle changes on the system. For instance, to provide the pausing capability, one of the debuggers' tactics is to set breakpoints. Using breakpoints is further assisted with either special hardware [26] (e.g., DR Registers of CPU and specific API to access/alter them), or software (e.g., specific opcodes) [98]

mechanisms. Single stepping, as another instance, is made possible by triggering exceptions in the code which is aided by a specific flag (trap flag). One way malware can discover the presence of the debugger is through the very same aiding mechanisms or results of corresponding changes on the system, which we will elaborate more as we proceed.

Based on the needed functionality, debuggers are implemented following different approaches [42], including user-mode debuggers (e.g., *OllyICE*, *OllyDbg*), kernel debuggers (e.g., *WinDbg*, KD), virtualization-based debuggers (e.g., *Ether* [31], *BOCHS*, [70], *HyperDBG* [38]), and more recently, the bare-metal debuggers such as MALT [148]. Each of the designs yields different levels of transparency. Kernel-level debuggers are more transparent than user-level debugger and provide more detailed information as they operate in ring 0 (same level of privilege as the OS). Virtualization/emulation-based debuggers have an even higher level of privilege than kernel debuggers since the OS in this setting is running atop the simulated (virtualized/emulated) hardware [31, 70] and consequently are more transparent to malware. Finally, the bare-metal designs such as MALT, which often rely on System Management Mode (SMM), offers the highest level of transparency against which many of the traditional anti-debugging techniques lose efficacy. In the next section, we proceed to manual dynamic analysis evasion, or more specifically, anti-debugging tactics.

3.2 Proposed Anti-Debugging Classification

We propose two major categories as malware's initial anti-debugging strategies, which are similar to the categories of automated analysis evasion: detection-dependent and detection-independent. We elaborate on each category down to the tactic level along with several representative techniques for each tactic. Table 1 captures a summary of our study on manual dynamic analysis evasion.

3.2.1 Detection-Dependent. We call the tactics of this category as *detection-dependent* in that the malware probes for detecting its environment in order to escape analysis. Put differently, a successful evasion is subjected to detecting or finding signs of an analysis environment. One of malware's major leverages against debuggers stems from the fact that debuggers were originally devised to debug legitimate software [110]. Thus, initially, there has been no stealthy countermeasure provisioned by them. Moreover, when we want to enable debugging capability, we often have to instrument the system with necessary tools which obviously results in a wide spectrum of traces in different levels of the system [24, 119]. Examining the system for such traces that are inherent to working principles of debuggers is one major strategy that malware utilizes to detect the presence of a debugger. In this section, we will elaborate on these system changes and how malware might try to disclose them. We would also briefly note how the defenders might try to counter such tactics.

Tactic 1. Fingerprinting. Fingerprinting is the malware's endeavor to spot, find, or detect signs that attest to the presence of an analysis environment or debuggers. Fingerprinting is the most common evasion tactic regarding both manual and automated analysis. However, the techniques are different. In contrast with the automated analysis evasion, the majority of fingerprinting techniques aimed at evading manual analysis involve fingerprinting the environment to detect the presence of debuggers [24]. Major fingerprinting techniques used by malware include the following:

— *Analyzing Process Environment Block (PEB). PEB* is a data structure that exists per process in the system and contains data about that process [90]. Different sections of *PEB* contain information that can be probed by malware to detect whether a debugger exists. The most

obvious one is a field inside PEB named *BeingDebugged* which can be read directly or as Microsoft recommends—and malware like *Kronos* [52] or *Satan RaaS* [129] implement—through the specific APIs that read this field, i.e., *IsDebuggerPresent()*, *CheckRemoteDebuggerPresent()*. Implementation of this technique is of trivial complexity which can be countered through alteration of *BeingDebugged* bit [102] or API hook [3]. Collectively, anti-debugging tactics relying on *PEB* constitute the majority of anti-debugging techniques observed in malware [17].

— *Search for Breakpoints.* To halt the execution, debuggers set breakpoints. This can be accomplished through hardware or software techniques. In hardware breakpoints, the breakpoint address, for instance, can be saved in CPU DR registers. In software breakpoints, the debugger writes the special opcode *0xCC* (*INT 3* instruction) into the process which is specifically designated for setting breakpoints. Consequently, the malware, if it spots signs of these breakpoints, presumes the presence of a debugger. This can be accomplished through a self-scan or integrity check, looking for *0xCC* or using *GetThreadContext* to check CPU register [40]. The latter technique has been employed by malware such as *CIH* [82] or *MyDoom* [84]. This technique is the second most observed anti-debugging technique in malware's arsenal [17] and is simple to implement (less than 10 lines of assembly code often suffices). Countering this category of tactics, however, is not trivial. In the case of software breakpoints, for instance, the debugger has to keep and feed a copy of the original byte that was replaced by *0xCC* opcode [102].

— *Probing For System Artifacts.* From installation to configuration and execution, debuggers leave traces behind in different levels of the OS (e.g., in the file system, registry, process name). Hence, malware can simply look for these traces. *FindWindow()* and *FindProcess()* are a couple of APIs that *shcndhss* [57] exploited to detect debuggers. The malware, for instance, can give the name of debuggers as the parameter to *FindWindow()* to verify if its process exists on the system or not [120, 131]. Most often, simple anti-debugging techniques are defeated with trivial complexity. The countermeasure to this category of tactic is randomizing the names or altering the results of the aforementioned query through simple API hooks. Even though attributed to one of the anti-debugging techniques in literature, we have rarely observed it in the wild. A similar technique to note here is called *parent check*. Ordinarily, applications are executed either through double clicking of an icon, or execution from command line, the parent process ID of which is retrievable accordingly. It would be a pronounced sign of debugger if the examined parent process name belongs to a debugger or is not equivalent to the process name of *explorer.exe* [40]. One straightforward technique is using CreateToolhelp32Snapshot() and checking if the parent process name matches the name of a known debugger [3]. A malware such as [57] uses such a technique [17]. Countering this technique would be skipping the relevant APIs [119]. Our observations demonstrate few utilizations of this technique [17]. Another technique for fingerprinting for system artifacts is mining *NtQuerySystemInformation*. The *ntdll NtQuerySystemInformation()* is a function that accepts a parameter which is the class of information to query [40]. Defeating this technique requires patching the kernel. *vti-rescan* [134], *Wdf01000.sys* [56], and *Inkasso trojaner* [113] are some instances that leverage this technique.

— *Timing-Based Detection.* Timing-based detection is among the most efficacious fingerprinting techniques for inferring a debugger's presence. Adroit employment of this technique reliably exposes the presence of a debugger and circumventing them is an arduous task. The logic behind this timing-based detection follows malware authors' presumption that a particular function or instruction set requires merely a minuscule amount of time. Thus, if a predefined threshold is surpassed, malware would infer the presence of a debugger or

analysis environment. Timing-based detection can be carried out either locally with the aid of local APIs (*GetTickCount()*, *QueryPerformanceCounter()*, etc. [89]) or CPU *rdtsc* (read timestamp counter), or can be performed by inquiring an external resource through the network [106] to conduct the timing. Local timing is simple to employ and difficult to circumvent. Countering timing-based detection conducted with the aid of an external resource (using NTP or tunneled NTP), however, is still an open problem [148]. *W32/HIV* [60], *W32/MyDoom* [84], and *W32/Ratos* [86] are infamous malware that are known to have exploited the timing discrepancies.

Tactic 2. Traps. Not to be mistaken with the trap flags, we define this category of tactics as "traps." Following this tactic, the malware provisions codes that when traversed or stepped through by debugger, production of specific information or lack thereof would help the malware infer the presence of a debugger. These inferences mostly rely on exploiting the logic of the system (e.g., *SEH* Exception handling [91]). Many techniques fit this category and we elaborate a couple of them here and hint at the rest in Table 1.

One technique by which malware beguiles a debugger to disclose cues of its presence is using specific instructions and exploiting the logic of how these instructions are handled. The handling is performed through Structured Exception Handling (*SEH*). (You can refer to [91] for more information about *SEH*.) For instance, Max++ malware [54] embeds "*int 2dh*" instruction within its code. According to *SEH*, when this instruction is executed, in a normal situation, i.e., absence of debugger, an exception is raised and malware can handle it via a try-catch structure. However, if a debugger is attached, this exception will be transferred to the debugger rather than the malware; the absence of expected exception is the logic that the malware entertains to deduce the presence of a debugger. Malware may employ other techniques to lay their traps such as embedding specific instruction prefixes [131], or other instructions such as 41h [40]. These techniques are of low complexity and can be implemented with less than 20 lines of code (in assembly). One way of countering these traps requires debuggers to skip these instructions. According to our survey, utilizing traps is a fairly common approach [17].

Tactic 3. Debugger Specific. Debugger specific evasion exploits vulnerabilities that are exclusive to a specific debugger. These vulnerabilities are difficult to discover, but simple to put into action. A famous instance pertains to *OllyDBG* [143]. Regular versions of this debugger have a format string bug which can be exploited to cause it to crash by passing an improper parameter to *OutPutDebugString()* function. Another pervasive-at-the-time technique was related to *SoftICE* debugger which was susceptible to multiple DoS attacks because of two vulnerable functions [39]. Malware like [19] that exploited these vulnerabilities would cause the *bluescreen of death*. *SoftICE* is no longer supported and the *dll* file that caused the vulnerability in *OllyDBG* is now fixed. But the idea still remains, if a vulnerability within a specific debugger is discovered, exploiting it would be a potent anti-debugging technique.

Tactic 4. Targeted. The last tactic of the detection-dependent category is targeted evasion. Through this tactic, the malware encrypts its malicious payload with a specific encryption key. This encryption key, however, is chosen to be an attribute or variable that could be found only on its target system. This key could be the serial number of a component in the target system, specific environment setting, and so forth. The targeted tactic is considered one of the most advanced analysis evasion tactics and it is effective against all kinds of debuggers from user mode to bare-metal. Countering the targeted tactic inherently is an arduous task and we will discuss why as we proceed. Following are two techniques for using the targeted tactic.

— *Environment Keying*. With this technique, the malware author encrypts the payload with a key that is possible to derive merely from the target environment. This could be a specific string in the registry or the serial number of a specific device. Gauss [14] and Ebowla (a framework) [76] are instances that use this technique to prevent their payloads from being analyzed. An obvious way to counter this technique is brute-forcing the payload to come up with the Encryption Key. And depending on the encryption algorithm and key complexity, it might not be feasible. Another glimpse of hope for combating this technique is brought by path exploration techniques such as [20, 27, 95, 136]. But these techniques have their own limits and may not be effective all the time. In fact, they lose their efficacy when facing the next generation of AI-powered targeted techniques.
— *AI-Powered Keying*. Attacks on Deep Neural Networks (DNNs) are on the rise, and so are cases of abusing them [75]. A recent instance relevant to this section involves IBM researchers who have come up with a proof-of-concept on how malware authors can benefit from AI to craft extremely evasive malware [30]. They developed DeepLocker [30], which encrypts its payload with an encryption key. The crucial difference, though, is that DeepLocker uses AI for the "trigger condition." Using the neural network, DeepLocker produces the key needed to decrypt the payload. This technique leverages the black-box nature of DNN to transform a simple *if this, then that* condition into a convolutional network. And due to the enormous complexity of neural networks, it becomes virtually impossible to exhaustively enumerate all possible pathways and conditions [29].

If crafted adroitly, the targeted tactic can be effective against all four types of debuggers.

3.2.2 Detection-Independent. Unlike the previous category, the aim of which was to detect or infer a debugger's presence or analysis environment, tactics of this category do not rely on the detection of their target system. They do not care whether the system on which they land has a debugger attached to it or not. The malware operates the same way regardless of their target system. We will have a survey on the tactics of this category in the following.

Tactic 1. Control Flow Manipulation. Through this tactic, malware exploits the implicit flow control mechanism conducted by Windows *OS*. To implement these techniques, malware authors often rely on callbacks, enumeration functions, thread local storage (*TLS*), and so forth [40, 119]. There are several noteworthy techniques here and each deserves a brief introduction.

— *Thread-Hiding*: A simple and effective technique is thread-hiding which, if used, prevents debugging events from reaching debugger. Microsoft has provisioned special APIs to this end [99, 135]. This technique uses the *NtSetInformationThread()* function to set the field *HideThreadFromDebugger()* of the*ETHREAD* kernel structure [143]. This is a powerful technique and simple to implement and can be countered by hooking the involving functions. *LockScreen* [135] is an instance known to have utilized this technique.
— *Suspending Threads*: A more aggressive way malware might step into is striving to halt the process of the debugger to continue its own execution with little trouble. This technique can be effective against only user-mode debuggers and can be carried out by leveraging *SuspendThread()* or *NtSuspendThread()* from *ntdll* [17]. Suspending threads is one of the anti-debugging techniques that *Kronos* banking malware used in its arsenal [52].
— *Multi-Threading*: Another technique to bypass debuggers and continue the execution is multi-threading. One way to implement this tactic is utilizing the *CreateThread()* API. A malware that is packed often spawns a separate thread within their process to perform the decryption routines [143]. However, there are instances where malware executes a part of its malicious code through a different thread outside the debugger. *McRat* [132] and

Vertexnet [28] have incorporated such a technique. Countering this technique is tricky. One way is to set breakpoints at every entry point [39, 93].

— *Self-Debugging*: Self-debugging is an interesting technique which prevents the debugger from successfully attaching to the malware [140]. By default, each process can be attached to merely one debugger. Malware such as *ZeroAccess* [128] exploits this by running a copy of itself and attaches to it as a debugger. Hence preventing another debugger to own it. There are several ways to implement this tactic: for instance, by leveraging *DbgUiDebugActiveProcess()* or *NtDebugActiveProcess()*.

Collectively, control-flow manipulation techniques are not very common among the samples we have observed.

Tactic 2. Lockout Evasion. In Lockout tactic, malware continues its execution by impeding with the working of the debugger without having to look for its presence. One way is to opt for the *BlockInput()* function as in the case of *Satan RaaS* [129], through which malware prevents mouse and keyboard inputs until its conditions are satisfied. Other techniques involve exploiting a feature in Windows NT-based platforms that allow the existence of multiple desktops. Malware such as *LockScreen* [135], with the help of *CreateDesktop()* followed by *SwitchDesktop()*, can select a different active desktop and continue its working unbeknownst to the debugger [39]. This tactic is mostly effective against traditional debuggers.

Tactic 3. Fileless Malware. Fileless malware, non-malware, and occasionally called Advanced Volatile Threats (*AVTs*), are among the latest trend in the evolution of malware [105, 118]. In contrast to all prior existence of malware, fileless malware requires no file to operate and they purely reside in memory and take advantage of existing system tools, e.g., *PowerShell* [80].

The purpose of such attacks is to make the forensics much harder. To analyze malware is to analyze its executable; and in fileless malware, there is no executable to begin with. In some cases such as SamSam [78], the only way to just retrieve a sample for analysis would be to catch the attack taking place live. These attacks inherently are not easy to conduct; but, with the help of exploit-kits are more readily available. A 2018 report by McAfee shows a 432% increase of fileless malware in 2017 [9] and projected to constitute 35% of attacks in 2018 [109]. Fileless tactic falls effective against all four types of debuggers.

With the aid of debuggers, the analyst can overcome many hurdles and limitations of static analysis. However, new trends and real-world scenarios in which the vendors face thousands of new malware samples daily demands a more agile approach—beyond the capabilities of manual dynamic analysis. In the next section, we discuss the emergence of automated dynamic analysis approach and Sandboxes as a response to these challenges and will further elaborate on malware's tactics to thwart them. Table 1 summarizes our survey of manual dynamic evasion techniques.

3.3 A Discussion

In Table 1, we summarize our survey on malware manual dynamic analysis evasion. In this table, we compare the techniques based on four criteria, i.e., complexity of implementation, resistance against manual dynamic analysis, pervasiveness, and efficacy level. In addition, in the example column, we provide malware samples that incorporate the corresponding techniques.

Manual dynamic analysis evasion or anti-debugging is as old as the existence of software. What we have observed in the last decade, attest that incorporation of anti-debugging is prevalent and has been steadily increasing. Authors in [22] demonstrate that on average, more than 70% of malware samples utilize anti-debugging techniques. The majority of these techniques are the ones we have covered in this section. In addition, we have noticed that besides the very pervasive techniques such as *IsDebuggerPresent*, malware are also employing more timing-based techniques.

Table 1. Classification and Comparison of Malware Anti-Debugging Techniques

Cat.	Tactic	Technique	Criteria	Complexity	Resistance	Countermeasure Tactic	Pervasiveness	Malware Sample	Efficacy -Level
Detection-Dependent	Fingerprinting	Reading PEB	IsDebuggerPresent() CheckRemoteDebuggerPresent()	Low	Low	Set the Beingdebugged flag to zero	Very high	[52, 129]	1
				Medium	Low	Set heap_groawable glag for flags field and forceflags to 0			1
			NtGlobalFlags()	Low	Medium	Attach debugger after process creation			1
		Detecting Breakpoints	Self-scan to spot INT 3 instruction Self-integrity-check	Low	Medium	Set breakpoint in the first byte of thread	High	[82, 84]	1, 2
			Read DR Registers (GetThreadContext() etc.)	Low	Medium	Reset the context_debug_registers flag in the contextflags before/after Original ntgetcontextthread function call			1, 2
		System Artifacts	FindWindow(), FindProcess(), FindFirstFile(),	Low-High	Low-High	Randomizing variables, achieve more transparency	Medium	[57]	1, 2, 3
		Mining NTQuery Object	ProcessDebugObjectHandle() ProcessDebugFlags() ProcessBasicInformation()	Medium	High	Modify process states after calling/skipping these API	Medium	[56, 113, 134]	1, 2
		Parent Check	GetCurrentProcessId() + CreateToolhelp32Snapshot()+ (Process32First())+Process32Next()	Medium	Medium	API hook	Low	[57]	1, 2
		Timing-Based Detection	Local Resource: RDTSC timeGetTime(), GetTickCount(), QueryPerformanceCounter GetLocalTime() GetSystemTime()	Low	High	Kernel patch to prevent access to rdtsc outside privilege mode, Maintain high-fidelity time source, Skip time-checking APIs	Medium	[60, 84, 86]	1, 2, 3, 4
			Query external time source (e.g. NTP)	Medium	N/A	None, open problem			
	Traps	Instruction Prefix (Rep)		High	Medium	Set breakpoint on exception handler, Allow single-step/breakpoint exceptions to be automatically passed to the exception handler	High	[54]	1, 2, 3
		Interrupt 3, 0x2D		Low	High				
		Interrupt 0x41		Low	High				
	Debugger Specific	OllyDBG: InputDebugString()		Low	High	Patch entry of kernel32!outputdebugstring()	Low	[19]	1, 2, 3
		SoftICE Interrupt 1		Low	High	Set breakpoint inside kernel32!createfilefilew()			
	Targeted	APT Environment Keying		High	Very High	Exhaustive Enumeration, path exploration techniques	Low	[14, 76]	1, 2, 3, 4
		AI Locksmithing		Very High	Very High	N/A	Rare	[30]	1, 2, 3, 4
Detection-Independent	Control Flow Manipulation	Self Debugging	DebugActiveProcess() DbgUiDebugActiveProcess() NtDebugActiveProcess()	Medium	Low	Set debug port to 0	Low	[128]	1, 2, 3
		Suspend Thread	SuspendThread() NtSuspendThread()	Low	Low	N/A		[52]	1, 2
		Thread Hiding	NtSetInformationThread() ZwSetInformationThread()	Low	Low	Skip the APIs		[135]	1, 2
		Multi-threading	CreateThread()	Medium	Low	Set breakpoint at every entry		[25, 135]	1, 2
	Lockout Evasion	BlockInput(), SwitchDesktop()		Low	Low	Skip APIs	Low	[129, 135]	1, 2, 3, 4
	Fileless (AVT)	Web-based exploits System-level exploits		High	Very High	N/A	Low	[36, 78]	1, 2, 3, 4

On the defensive side, we note two major approaches. First is the reactive approach which tries to equip the debugger with counter evasion with respect to any evasion vector such as [119, 133]. Second are endeavors focused on building more transparent analysis systems such as *Ether*, *HyperDBG*, and *MALT* [38, 148]. These methods significantly raise the bar for the traditional fingerprinting tactic. But, the more novel and advanced targeted techniques can foil their efforts.

Another emerging trend in malware industry is fileless malware. Such malware as in the case of *Rozena* [45] leverages exploits to execute malicious commands right from the memory. Given

the difficulties of acquiring samples in post-infection phase, perhaps we can conclude that fileless (*AVT*) tactics along with the targeted, are the most effective manual dynamic analysis evasion tactics.

4 AUTOMATED DYNAMIC ANALYSIS EVASION

Although effective, the manual dynamic analysis suffers a critical limitation, that is, time. 2018 statistics provided by *McAfee* reports on receiving more than 600K new samples each day [68]. Analyzing this massive number of malware samples calls for a far more agile approach. This demand led to a new paradigm of analysis which we referred to as automated dynamic analysis. Sandbox is the representative technology for this paradigm. In this section, we will have a brief introduction to sandboxes and further propose a classification and comparison of malware evasion tactics.

4.1 An Overview of Malware Sandboxes

The concept behind a malware automated dynamic analysis system is to capture the suspicious program in a controlled and contained testing environment called sandbox, where its behavior in runtime can be closely studied and analyzed. Initially, sandboxes were employed as a part of the manual malware analysis. But today, they are playing their roles as the core of the automated detection process [66]. Sandboxes are built in different ways. To better grasp the evasion tactics, and depict how they stand against different sandbox technologies, first, we must have a sense of how they are made.

4.1.1 Virtualization-Based Sandboxes. A virtual machine (VM) according to Goldberg [47], is "an efficient, isolated duplicate of the real machine." The hypervisor or virtual machine manager (*VMM*) is in charge of managing and mediating programs' access requests to the underlying hardware. In other words, every virtual machine atop the *VMM*, in order to access the hardware, must first pass through the hypervisor. There are a couple of ways to implement sandboxes based on virtualization. One way is to weave the analysis tools directly into the hypervisor as in the case of Ether [31]. The other approach would be to embed the analysis tools (e.g., installing hooks) within the virtual machine that runs the malware sample. Instances of this design are *Norman* sandbox [101], *CWsandbox* [137], and more recently *Cuckoo* sandbox [49]. Both of the methods inherently leak subtle cues which malware could pick on to detect the presence of a sandbox. In the latter case, for instance, the *VMM* has to provide the required information to the analysis VM, which means whenever a sensitive system call is being made by the malware, the *VMM* has to pass the control to the analysis tools inside the VM. Challenges of performing these procedures without leakage are profound, which we will elaborate accordingly.

4.1.2 Emulation-Based Sandboxes. An emulator is a software that simulates a functionality or a piece of hardware [35]. An emulation-based sandbox can be achieved through different designs. One would be to simulate the necessary OS functions and APIs. Another approach is the simulation of CPU and memory and is the case for many anti-virus products [35]. Simulation of I/O in addition to memory and CPU is what in literature is referred to as the full system simulation. *QEMU* [12] is a widely famous full system simulation based on which other famous sandboxes such as *Anubis* [10] are built. Among the eminent features of emulation-based sandboxes are the great flexibility and detailed visibility of malware inner workings (introspection) that they offer. Especially with the full system emulation, the behavior of the program under inspection (PUI) could be studied with minute details.

4.1.3 Bare-Metal Sandboxes. Recently evolved and perplexing evasion tactics, employed by sophisticated malware, demand a new paradigm of analysis. The emerging idea is to execute the malware in several different analysis environments simultaneously with the assumption that any deviation in behavior is a potential indication of malicious intent [63]. The feasibility of this idea requires a reference system in which the malware is analyzed without the utilization of any detectable component, and the ideal choice would be a bare-metal environment equal to a real production system in terms of transparency. There have been several products in this vein (e.g., *Bare-box*, bare cloud [62, 63, 138]). Along with the merits that each design offers, there are subtle flaws or specific working principles that malware exploits to forge their evasion tactic.

4.2 Proposed Classification of Automated Dynamic Analysis Evasion Techniques

If the malware achieves one specific goal, it can triumphantly evade the sandbox. This goal is to behave nicely or refrain from executing its malicious payload so long as it resides within the sandbox. This strategy capitalizes on two facts. The first is that due to a massive number of malware samples and limitation of resources, sandboxes allot a specific limited time to the analysis of a sample. The second stems from an inherent limitation of dynamic analysis. In dynamic analysis, since the "runtime behavior" and "execution" are being inspected, only the execution path is visible to the inspector (sandbox). Thus, if a malware portrays no malevolent behavior while under examination, the sandbox flags it as benign.

Similar to manual dynamic analysis evasion, we propose to classify automated dynamic analysis evasion tactics under two categories, i.e., detection-dependent evasion and detection-independent evasion. We will elaborate on these tactics along with several techniques under each tactic. In addition, we briefly note the ways through which sandboxes try to defeat these tactics. To provide a more coherent overview of these evasion techniques, we employ the three following criteria which would serve as the basis for our comparison and we summarize our findings in Table 2.

- **Complexity:** The difficulty of implementing an evasion tactic. Our touchstones for evaluating complexity are lines of code (*LOC*), and more importantly the challenges which differ per tactic.
- **Pervasiveness:** This criterion captures the relative prevalence of a particular tactic. Based on our observations or security reports [68, 85], we try to provide a relative prevalence for each tactic.
- **Efficacy Level:** Each tactic has an efficacy level. This demonstrates against which type of sandbox that particular tactic is effective.

4.2.1 Detection-Dependent Evasion. Similar to the corresponding category of evasion in the manual analysis evasion, the main goal of the malware is to detect its environment to verify whether the host is a sandbox or not. A successful evasion in this category is subjected to correctly detecting the environment. If the environment is detected to be a sandbox, or that it is not the intended environment, the malware will not reveal its malicious payload, hence, evades the detection. Following are the tactics a malware might use to achieve this.

Tactic 1. Fingerprinting. Fingerprinting is a tactic pursued by malware to detect the presence of sandboxes by looking for environmental artifacts or signs that could reveal the indications of a virtual/emulated machine. These signs can range from device drivers, overt files on disk and registry keys, to discrepancies emulated/virtualized processors. Indications of *VM* or emulation are scattered at different levels [51, 59, 69, 112]. It is noteworthy to mention that initially, many sandboxes such as Norman [101] were developed upon *VMs*. Thus, in literature, you may still observe the terms sandbox and *VMs* being used interchangeably to imply a contained analysis

Table 2. Classification and Comparison of Malware Sandbox Evasion Techniques

Criteria		Complexity	Pervasiveness	Efficacy Level	Sandbox Countermeasure Tactics		Detection Complexity	Example
Cat.	Tactic				Complexity	Effectiveness		
Detection-Independent	**Stalling**	Low-Medium	Medium	All Architectures	Sleep Patching		Very High	[13, 14, 77, 79]
					Low	Low		
	Trigger-Based	Low	Medium	Emulation-Based, Bare-metal	Path Exploration		Moderate	[17, 25, 83, 84, 114, 126]
					High	Moderate		
	Fileless (AVT)	High	Low	All Architectures	N/A		Very High	[78]
Detection-Dependent	**Fingerprinting**	High	High	VM-Based, Hypervisor-based, Emulation-Based	Using heterogeneous analysis, Artifact Randomization		Moderate	[2, 65, 87, 88, 124, 127, 139, 145]
					Moderate	Moderate		
	Reverse Turing Test	Medium	Medium	VM-Based, Hypervisor-based, Bare-metal,	Digital Simulation, path exploration		Moderate	[53, 142]
					Low	Low		
	Targeted	Very High	Low	VM-Based, Hypervisor-based, Emulation-based	Path exploration		Very High	[37, 48, 101]
					High	Low		

environment. The very same fact is also the reason for the advent of techniques referred to as anti-VM, suggesting that detection of a virtual machine would potentially mean an analysis environment. Different studies have been conducted to uncover the levels at which sandboxes leave their marks [24, 144]. These levels are as follows:

- **Hardware:** Devices and drivers are overt artifacts that malware might look for to identify its environment. In the case of devices, *VMs* often emulate devices that can be readily detected as in the case of Reptile malware [2]. This ranges from obvious footprints such as the *VMWare* Ethernet device with its identifiable manufacturer prefix, to more subtle marks. Moreover, specific drivers are employed by VMs to interact properly with the host OS. These drivers are other indications of an analysis environment for malware. For instance, in the path `C:\Windows\System32\Drivers` exist such signs that could expose *VMWare*, *VirtualBox*, and so forth (e.g., *Vmmouse.sys*, *vm3dgl.dll*, *VMToolsHook.dll*) [4].
- **Execution Environment:** A malware inside sandbox experiences subtle differences in the environment within which they are executed. Kernel space memory values, for instance, are different between a sandbox and native system that can be detected by malware as in the cases of *Agobot* and *Storm* Trojans [124, 127]. Artifacts of this level manifest themselves either in memory or execution. As in the case of memory artifacts, for instance, to allow inspection and control between host and guest OS, *VMWare* creates a channel between them ("*ComCHannel*"). Virtual PC hooks work in the same way [67]. Anticuckoo [34], for instance, detects and crashes the cuckoo sandbox through detecting and manipulating the system hooks.
- **Application:** When an analysis application is used in an environment, its presence is usually disclosed due to the artifacts of installation and execution which can be picked up by inquisitive malware such as *Rbot* [88], *Phatbot* [87], [139], or [65]. Even if not executed, evidence of the analysis tools might be residing on the disc, registry keys, and so forth, which could be readily found by malware, especially if they contain a well-known file name or locations, e.g., `SYSTEM\CurrentControlSet\Control\VirtualDeviceDrivers`. In a similar instance, if the names are not altered or the corresponding processes concealed, they can be enumerated by malware with little trouble. For instance, *VMtools.exe*, *Vmwareuser.exe*, or

vboxservice.exe if queried by malware, are vivid indicators of a virtual machine and analysis environment.

— **Behavior:** Perhaps, the most troublesome of all, for anti-malware vendors to conceal, are the leakages caused by imperfect virtualization/emulation, or characteristics that are innate to such environments. Of particular interest are discrepancies of behavior between an emulated CPU and a physical one [112]. Performance of an application running under emulation is inferior to a real system. This performance penalty stems from translations or interceptions that have to be carried out in an emulation/virtualization. Such discrepancies can be disclosed through a diverse set of timing attacks [41, 44, 106]. Computing absolute performance for fingerprinting is an arduous task due to the diversity of hardware configuration.

  An alternative for malware is to calculate the relative performance [112]. Pursuing this method, malware compares the performance ratio of two or more operations on the same system. If the measurement varies significantly among production and emulation systems, there's a high chance of sandbox presence. Another interesting way of exploiting the limitations of emulation is observing and comparing the effects of caching on emulated and real environments. In this technique, a function is executed a number of times. As expected, the first run must be the slowest. The same test is performed again, this time, in the absence of caching. Timing analysis depicts the effectiveness of the employed caching. Simulation of processor cache is a complex task and emulators may not support it, the result of which is exposure of emulation in the aforementioned test [112]. Another noteworthy set of techniques are the red pill tests [104]. An inevitable behavioral artifact sandboxes share, are the imperfect simulation of CPU and residing instruction bugs. If a malware specifically finds such bugs and if during the execution it notices a mishandled instruction, it will suspect the presence of the sandbox.

— **Network:** In addition to the previous levels that were suggested by Chen et al. [24], malware can also probe the network in pursuit of sandbox's marks. These marks are manifested in many forms such as known fixed IP addresses [145], limitations pertaining to the sandboxes that prevent or emulate the internet access [15, 125], or extremely fast internet connection [33]. One technique, for instance, proposed in [145], is detecting sandboxes based on their known IP addresss, which is acquired in an earlier attack through a decoy malware.

Fingerprinting tactic was the initial endeavor of malware authors to detect and evade sandboxes, and as we observed the techniques have evolved significantly. In countering fingerprinting, most solutions are reactive, namely, the fingerprinting technique must first be disclosed, then the corresponding counter evasion will be provisioned. A sound offensive way of fighting against fingerprinting are endeavors such as Jing et al.'s *Morpheus*. *Morpheus* is a tool that *automatically* finds fingerprints on QEMU and VirtualbBox-based emulators [21]. Collectively, the fingerprinting tactic is still the dominant approach to detect and evade sandboxes [85].

Tactic 2. Reverse Turing Test. The second tactic that aims at detection, is checking for human interaction with the system. This tactic capitalizes on the fact that sandboxes are automated machines with no human or operator directly interacting with them. Thus, if malware does not observe any human interaction, it presumes to be in a sandbox. Such tactic is referred to as Reverse Turing Test since a machine is trying to distinguish between human or AI. This tactic can be carried out through various techniques [33, 37, 59, 122, 142]. For instance, the *UpClicker* [142] or a more advanced one, *BaneChant* [53], await the mouse left-click to detect human interaction [116]. In a large portion of reverse Turing test techniques, the malware looks for last user's inputs. To do so, malware often leverages a combination of *GetTickCount()* and *GetLastInputInfo()* functions

to compute the idle time of the user. To test whether it is running on a real system, the malware waits indefinitely for any form of user input. On a real system, eventually a key would be pressed or mouse would be moved by the user. If that occurs for a specific number of times, malware executes its malicious payload. To counter this tactic of evasion, simulating human behavior might seem intuitive but it might be counterproductive. One reason is that digitally generated human behavior can also be detected [123]. The second reason is that limitation of designing such reversal Turing test seems to be merely a function of imagination. To demonstrate our point, consider another technique in which the malware waits for the user to scroll to the second page of a Rich Text Format (RTF) before it executes the malicious payload; or in another approach, using Windows function *GetCursorPos()*, which holds the position of the system's cursor, the malware checks the cursor movement speed between instructions; if it exceeds a specific threshold, it would imply that the movement is too fast to be human-generated and malware ceases to operate [116]; or another more recent technique which relies on seeking wear and tear signs of a production system [92]. The seemingly endless possibilities for designing reversal Turing tests make it a puzzling challenge for sandboxes to counter. This tactic is becoming more prevalent, but still not as pervasive as fingerprinting [85].

Tactic 3. Targeted. The third tactic of the detection-dependent category is targeted detection. This tactic is slightly different from the previous ones in that instead of striving for detecting or evading a sandbox directly, the malware fingerprints the environment to verify if the host is precisely the intended (targeted) machine. In other words, the malware looks for its target, not sandbox. This tactic can be employed following different routes.

- **Environmentally Targeted:** *Stuxnet* is known to be the first Cyber weapon that incorporated targeting tactic as a portion of its evasion tactics [37]. *Stuxnet* explicitly looked for the presence of a specific industrial control system and would remain dormant otherwise. Depending on the infection approach, APT attacks follow different routes.
- **Individually Targeted:** In the case of *Stuxnet*, the strategy was to keep probing victims (a wormy behavior) until the target was reached. Other classes of APT include the attacks such as *darkhotel* [48], in which the infection is conducted through spear phishing (i.e., directly aimed at the target). In other words, the attackers make sure that their malicious code is directly delivered to their target.
- **Environment-Dependent Encryption:** Such targeted malware have an encrypted payload, the decryption of which is subjected to a key that is derived from its victim's environment. The key might be a hardware serial number, specific environmental settings, and so forth. You can refer to [94] for a thorough discussion of the topic.

4.2.2 Detection-Independent Evasion. The major difference of this category of tactics is that they do not rely on detecting the target environment, and their evasion tactic is independent of target system, which relieves them from having to employ sophisticated detection techniques. Consequently, efforts directed at achieving more transparency have no effect on these tactics. In the following, we elaborate on the tactics of this category and several techniques to deploy them.

Tactic 1. Stalling. This tactic of analysis evasion capitalizes on the fact that sandboxes allot a limited amount of time to the analysis of each sample. Malware has to simply postpone its malicious activity to the post-analysis stage [64]. To this end, malware authors have come up with the idea of stalling [67], which can be achieved through a diverse set of techniques that range from a simple call to *sleep()* function to more sophisticated ones. Here we examine some of the known major stalling techniques.

— **Simple Sleep:** Sleeping is the simplest form of stalling and just as simple to defeat. The idea was to remain inactive for *n* minutes in order for the sandbox inspection to timeout before observing any malicious activity. After being released to the network, the malware would execute the malicious payload. The infamous *DUQU* [14] exhibits such technique as one of its prerequisites for ignition. It requires that the system remain idle for at least 10 minutes [117]. Another example would be the *Khelios* botnet [79]. A new variant of the *Khelios* sample found in 2013 (Called Nap) calls the *SleepEx()* API with a timeout of 10 minutes. This delay in execution outruns the sandbox analysis timeout, which is followed by achieving the harmless flag. To counter such techniques, sandboxes came up with the simple idea of accelerating the time (called sleep patching). Although seemingly logical, sleep patching has unpredicted effects. An interesting side effect of sleep patching was observed in specific malware samples where the acceleration actually leads to inactivity of malware instances that wait for human interaction within a predefined period of time [123]. Despite its side effects, sleep patching turned out to be effective in some cases and malware authors came up with more advanced techniques as a response.
— **Advanced Sleep:** New malware instances such as *Pafish* [77] were found to opt for more advanced sleeping tricks. They detect sleep-patching using the *rdtsc* instruction in combination with Sleep() to check the acceleration of execution.
— **Code Stalling:** While the *sleeps* delay the execution, in code stalling the malware opts for executing irrelevant and time-consuming benign instructions to avoid raising any suspicions from the sandbox [123]. *Rombertik* [13] is a spyware aimed at stealing confidential data. To confuse sandboxes, it writes approximately 960 million bytes of random data to memory. The hurdles this technique impose on sandboxes are twofold. The first one is the inability of the sandbox to suspect stalling as the sample is running actively. The second one is that this excessive writing would overwhelm the tracking tools [13]. In another striking code stalling technique, the malware encrypts the payload using a weak encryption key and brute-forces it during the execution [63].

Tactic 2. Trigger-Based. The second tactic that does not rely on detection is a trigger-based tactic, or more traditionally logic bombs. As we have already noted, the basis of evading sandboxes is to simply refrain from exhibiting the malicious behavior so long as they are in the sandbox. Another tactic for remaining dormant would be to wait for a trigger. There are many environmental variables that can serve as malware triggers ranging from system date to a specially crafted network instruction. For instance, *MyDoom* [84] is triggered on specific dates and performs *DDoS* attacks, or some key-loggers only log keystrokes for particular websites; and finally, *DDoS* zombies, which are only activated when given the proper command [8]. In the literature, this behavior is referred to as trigger-based behavior [20]. In the following, there are a number of triggers embedded within malware as a protection layer against sandboxes.

— **Keystroke-Based:** The malware gets triggered if it notices a specific keyword, for instance, the name of an app or the title of a window [114]. What occurs when the trigger happens, depends on the purpose and context. For example, malware might initiate logging keystrokes.
— **System Time:** In this case, the system date or time would serve as the trigger [83, 114, 126]. A newer malware that recently used this technique was the industroyer [25].
— **Network Inputs:** A portion of malware are triggered when they receive certain inputs from the network as in the case of Tribal flood network [17].
— **Covert Trigger-Based:** Previous techniques often presuppose lack of code inspection to remain undetected (as is often the case for compiled malware). However, there have been advances regarding the automatic detection of such program routes, e.g., state-of-the-art

covert trigger-based techniques are being devised as a response to automated approaches that aim at detecting such triggers instruction. These techniques utilize instruction-level stenography to hide the malicious code from the disassemblers. In addition, they implement trigger-based bugs to provision stealthy control transfer, which makes it difficult for dynamic analysis to discover proper triggers [37]; return values from system calls are other instances of malware triggers.

When initially seen in the wild, trigger-based tactics were not utilized to evade sandboxes. They are still relatively seen in the wild and are enhanced to circumvent sandboxes. Finding these triggers is often pursued through path exploration approaches, e.g., symbolic execution with the goal of finding and traversing all the conditional branches in an automated manner. These approaches require significant levels of resources and are still below a fair level of efficiency.

Tactic 3. Fileless Malware. In Section 3.2.2, we discussed fileless malware. In addition to the rigorous resistance against manual analysis, fileless malware is profoundly adept at evading security mechanisms. A major difference of this tactic is that most often the malware is not subjected to the analysis environment in the first place. This is an inherent outcome of this tactic. The techniques that fileless malware utilizes are similar to drive-by attacks [71]. Generally, fileless malware exploits a vulnerability of the target system (OS, Browser, Browser plugins, etc.) and injects its malicious code directly into the memory. The new trend of fileless malware uses windows *PowerShell* to carry out its task. As mentioned earlier, this attack is on the rise, and defending against it is of great complexity.

4.3 A Brief Survey on Countering Malware Evasion

Countering evasive malware can follow different tactics. In this section, we briefly survey such defensive tactics against evasive malware. Table 3 summarizes our survey on countermeasure tactics against evasive malware.

4.3.1 Reactive Detection. In reactive detection, defenders tap into their knowledge of specific evasive behaviors to craft their countermeasure accordingly. Namely, the detection is subjected to knowing the evasion technique in advance. Lau et al. [69] pioneered this approach by devising DSD-TRACER which was later followed by other works such as [18]. In [18], for instance, Brengel et al. look for excessive usage of *rdtsc* and *CPUID*, which are utilized by malware such as *Industroyer* [25] to conduct timing attacks for fingerprinting and evasion. Reactive approach could circumvent only the known tactics and is vulnerable to novel evasion techniques. They are relatively easy to implement.

4.3.2 Multi-System Execution. In [21], the authors define the Multi-System Execution as a major counter evasion tactic. In this approach, malware is executed on several different analysis platforms. Their execution then is studied to determine if their behavior has been diverged based on the execution environment. In [6], for instance, Balzarotti et al., using a kernel driver, compare the system calls made in different environments to detect the divergence. In a similar tactic, Kirat and Vigna introduced *MALGENE* [61] in which they utilize a DNA protein alignment algorithm, to devise a new system called trace alignment technique. They executed 2,810 evasive malware in Anubis and Ether and managed to automatically generate *evasion signatures* (a combination of user API and system call events), and group them into 78 similar evasion techniques. There have been other proposals such as [58, 63, 74] that embody the multi-system execution tactic. Manifesting different behavior in different environments is the characteristic of a detection-dependent strategy. Thus, multi-system execution tactic is only effective against detection-dependent evasion strategy.

Table 3. Classification and Comparison of Countermeasure Tactics Against Evasive Malware

Criterion / Countermeasure	Effective Against	Complexity	Weakness	Examples
Reactive	Only Known Evasion techniques	Low	Vulnerable to zero-day techniques.	[18, 69, 133]
Multi-System Execution	Detection-dependent category	Medium	Ineffective against detection-independent tactic.	[6, 61, 58, 63, 74]
Path-Exploration	All tactics except Fileless	High	Vulnerable to anti-symbolic execution obfuscation. Not scalable, resource intensive.	[20, 95, 108]
Towards Perfect Transparency	Fingerprinting Tactic	Very High	Vulnerable to Targeted, Reverse Turing, Stalling, Fileless, and trigger-based tactics. Unless the sandbox is equipped with other countermeasure tactics as well.	[31, 63, 73, 100, 137, 149]

4.3.3 Path Exploration. Path exploration tactic as demonstrated in [20, 108] converges the focus on triggering as many conditional branches as possible in a program in order to provide a more thorough code coverage. The goal is to trigger the malicious payload and expose the malware. In [95], authors use the path-exploration tactic to identify the detection-dependent malware. Another seminal work under this tactic is Brumley et al.'s [20] MineSweeper in which they use symbolic and concrete execution to solve input values that trigger execution down each path.

Although the path-exploration tactic might seem to be the holy grail of exposing evasive malware, we must note that malware authors have weapons in their arsenal to combat them. For instance, malware authors can incorporate anti-symbolic execution obfuscation [7] into their code and impede with the path-exploration tactic. Or, the solver may fall into a corrupted path leading to crashes.

4.3.4 Toward Perfect Transparency. Research efforts for finding solutions to counter evasive malware, by a drastically wide margin, are focused on achieving more transparent systems. Transparent analysis systems, in effect, are a direct response to the proliferation fingerprinting tactic. We point out several key approaches to achieve transparency and hiding analysis systems from the prying eyes of malware.

- **Hiding Environmental Artifacts:** CWSandbox [137] was an early work on automated malware analysis which is based on *VMWare*. To prevent malware from detecting the analysis environment, authors instrument the system with *rootkit-like* functionality to cover the environmental artifacts (e.g., files, registry entries).
- **Hypervisor-Based Analysis:** Given the higher privilege in hypervisors, researchers have proposed to take their instrumentation of the system into the hypervisor itself. As noted earlier, authors in [31] proposed ether in which the activity of malware is monitored by catching context switches and system calls using Xen. However, in [106] the authors showed that this system is still vulnerable to fingerprinting tactic. More specifically, the timing-based detection. Other works which embody this defensive tactic are that of LenLengyel et al.'s *DRAKVUF* [73] and Nguyen et al.'s *MAVMM* [100], which are built with the aid of *Xen*, and *AMD SVMP*, respectively. Though the experimental results of such design might seem promising, they are shown to be detectable as well. Regardless, such systems are shown to be detectable as well [18, 106, 107, 130].
- **Bare-Metal Analysis:** The bare-metal analysis approach is the most idealistic endeavor which targets perfect transparency. In *BareCloud* [63], for instance, authors refrain from incorporating in-system monitoring tools. Rather, they rely on analyzing disc and network activity at the hardware level. They achieve a high level of transparency. However, at the

cost of introspection. If a malware opts for a stalling tactic, for instance, *BareCloud* would fail to detect the suspicious activity. Recently, the new trend of making a bare-metal system seems to be concerning the incorporation of system management mode (SMM). Specifically, several traits of SMM make them lucrative choices for building bare-metal systems including full access to system memory, inability to be modified once initialized, and blazing fast speed [149]. In [149], the authors introduce *Spectre* for transparent malware analysis. Zhang et al. further employed this technique to introduce *MALT*—a transparent debugger.

4.4 Discussion

In Table 2, we summarized our survey on malware automated dynamic analysis evasion. In this table, we compared the techniques based on four criteria, i.e., complexity of implementation, pervasiveness, efficacy level, and detection complexity. In addition, in the example column, we provided malware samples that incorporate the corresponding techniques. Moreover, under Sandbox countermeasure tactics, we hint on how the defensive side strives to counter the evasion tactics and how complex/effective these countermeasures are.

In this section, we provided our survey and classification on malware automated dynamic analysis evasion. Our observations, in line with studies such as [22, 85, 103], notes several trends.

Fingerprinting by far is the most pervasive tactic and has proven to be an effective technique if executed with precision. Aside from other researches and our observations, another proof regarding the dominance of fingerprinting tactic is the trend of defensive research in the last decade and how they are evolving [20, 31, 62, 63, 72, 95, 100, 115, 125, 133, 141, 147–149]. Examining these researches we observe that the great majority of them are pursuing more transparent systems, the reason of which is to be more resilient against fingerprinting and more broadly, detection-dependent tactics.

We see two reasons behind the pervasiveness of fingerprinting tactic. The first reason is the immensity of possibilities for fingerprinting scenarios. In each level of the system (hardware, application, network, etc.) there are always new hints of identity that malware can look for.

The second reason is the countermeasure strategies against the fingerprinting tactic. As we noted in the previous section, there are two major strategies in this regard. The first one is the reactive approach. Namely, for the sandbox to neutralize malware attempts directed at fingerprinting, it has to know what constitutes fingerprinting in the first place. The second strategy is to aim for more transparent systems either following the approach of single-stand analysis systems, e.g., [31] or heterogeneous analysis, i.e., bare-metal analysis [62, 63]. In retrospect to the cases we cited in this section, we know that both strategies are still vulnerable to zero-day fingerprinting techniques.

Another challenge of transparent systems is the reverse Turing test tactic employed by malware which virtually foils most of the efforts. Reverse Turing test tactic again, forces the defenders to defend with the reactive approach.

On the other hand, we are noticing new trends of malware attack employing detection-independent tactics. Most notably, stalling [103] and fileless tactics [9]. The majority of the efforts that have been put into developing more transparent systems, has raised the bar for detection-dependent tactics. That is one reason that has motivated malware authors to adopt more detection-independent tactics.

Unbound fingerprinting possibilities, reactive approaches, the vulnerability of transparent systems to zero-day detection-dependent tactics (fingerprinting, targeted, reverse Turing), and vulnerability of transparency-seeking strategies to detection-independent tactics, all seem to call for more generic and effective defensive strategies.

A promising approach that can potentially counter most of the evasion tactics is path exploration techniques. Every malware will expose its malicious payload if its specific conditions are satisfied. Path exploration techniques potentially can trigger those conditions and expose the malicious behavior. This promising approach, however, is also being shadowed by the emergence of AI-Based malware, which instead of a single *if-then* condition, uses a complex neural network.

5 CONCLUSION AND FUTURE DIRECTIONS

In this article, we conducted a survey on the subject of malware analysis evasion techniques and proposed classifications for both modes of manual and automated analysis.

We defined two major categories of evasion for both manual and automated analysis: detection-dependent and detection-independent. On the defensive side, defenders are pursuing four major strategies. Reactive approaches, more transparent analysis systems, bare-metal analysis, and several endeavors toward more generic approaches, namely, path exploration; the latter of which mostly concerns evasion tactics tailored to automated dynamic analysis evasion. The first three defensive strategies can hope for being effective merely against detection-dependent evasion tactics and lose their effectiveness against detection-independent tactics. As for the detection-independent tactic, we need more generic approaches such as the path exploration methods.

Parallel to endeavors toward path exploration methods, we advocate automated fingerprinting generations such as MORPHEUS [55] and Red Pills [104]. These solutions would not be a generic response, but they would significantly raise the bar for malware pursuing fingerprinting tactic.

Another line of inquiry that we suggest is analyzing the effectiveness of using malware evasion tactics against itself; for the case of detection-dependent tactics, a malware strives relentlessly to detect a sandbox and refrain from execution. What would be the result of a program installed on a production system that would impersonate a sandbox? How efficacious would it be in deterring malware, and if executed adroitly, and how significantly would such approach raise the bar for devising future evasion attempts?

ACKNOWLEDGMENTS

We would like to expand our gratitude to Professor Eric Filiol of ESIEA (C + V)O lab for his support and insights on this article.

REFERENCES

[1] Sanjeev Kumar Aggarwal and Sarath M. Kumar. 2002. Debuggers for programming languages. In *The Compiler Design Handbook*. CRC Press, 297–329.

[2] Anish. 2012. Reptile Malware - Behavioral Analysis. Retrieved October 2018 from http://malwarecrypt.blogspot.com/2012/01/reptile-malware-behavioral-analysis.html.

[3] Apriorit. 2016. Anti Debugging Protection Techniques. Retrieved October 2018 from https://www.apriorit.com/dev-blog/367-anti-reverse-engineering-protection-techniques-to-use-before-releasing-software.

[4] Yaniv Assor. 2016. Anti-VM and Anti-Sandbox Explained. Retrieved September 2018 from https://www.cyberbit.com/blog/endpoint-security/anti-vm-and-anti-sandbox-explained/.

[5] AV-TEST. 2017. The AV-TEST Security Report 2016/2017. Retrieved September 2018 from https://www.av-test.org/en/news/the-it-security-status-at-a-glance-the-av-test-security-report-20162017/.

[6] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2010. Efficient detection of split personalities in malware. In *Annual Network and Distributed System Security Symposium (NDSS)*. http://www.isoc.org/isoc/conferences/ndss/10/pdf/24.pdf.

[7] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 189–200.

[8] Jason Barlow. 2000. Tribe Flood Network 2000 (TFN2K). Retrieved September 2018 from https://packetstormsecurity.com/distributed/TFN2k_Analysis-1.3.txt.

[9] Alex Bassett, Christiaan Beek, Niamh Minihane, Eric Peterson, Raj Samani, Craig Schmugar, ReseAnne Sims, Dan Sommer, and Bing Sun. 2018. *MacAfee Labs Threat Report March 2018.* Technical Report. McAfee Labs. https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2018.pdf.

[10] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. 2006. *TTAnalyze: A Tool for Analyzing Malware.*

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

[12] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, Vol. 41. 46.

[13] Alex Chiu Ben Baker. 2015. Threat Spotlight: Rombertik, Gazing Past the Smoke, Mirrors, and Trapdoors. Retrieved November 2018 from https://blogs.cisco.com/security/talos/rombertik.

[14] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi. 2012. The Cousins of Stuxnet: Duqu, Flame, and Gauss. *Future Internet* 4 (4), 971–1003.

[15] Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bulent Yener. 2016. AVLeak: Fingerprinting antivirus emulators through black-box testing. In *Proceedings of the 10th USENIX Conference on Offensive Technologies*. USENIX Association, 91–105.

[16] Jean-Marie Borello and Ludovic Me. 2008. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology* 4, 3 (2008), 211–220.

[17] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. 2012. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat* (2012).

[18] Michael Brengel, Michael Backes, and Christian Rossow. 2016. Detecting hardware-assisted virtualization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 207–227.

[19] Nicolas Brulez. 2012. Scan of the Month 33: Anti Reverse Engineering Uncovered. Retrieved October 2018 from http://old.honeynet.org/scans/scan33/nico/index.html.

[20] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 65–88.

[21] Alexei Bulazel and Bülent Yener. 2017. 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*. ACM, 2.

[22] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. 2016. Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware. In *IFIP International Information Security and Privacy Conference*. Springer, 323–336.

[23] Thomas M. Chen and Jean-Marc Robert. 2004. The evolution of viruses and worms. *Statistical Methods in Computer Security* 1 (2004).

[24] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC*. IEEE, 177–186.

[25] Anton Cherepanov. 2017. WIN32/INDUSTROYER, A New Threat for Industrial Control Systems. Retrieved October 2018 from https://www.welivesecurity.com/wp-content/uploads/2017/06/Win32_Industroyer.pdf.

[26] Michael Chourdakis. 2008. Toggle Hardware Data/Read/Execute Breakpoints Programmatically. Retrieved February 2018 from https://www.codeproject.com/Articles/28071/Toggle-hardware-data-read-execute-breakpoints-prog.

[27] Jedidiah R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhendong Su, S. Felix Wu, and Frederic T. Chong. 2006. Temporal search: Detecting hidden malware timebombs with virtual machines. In *ACM SIGARCH Computer Architecture News*, Vol. 34. ACM, 25–36.

[28] CTurt. 2012. Reverse Engineering VertexNet Malware. Retrieved March 2018 from https://cturt.github.io/vertex-net.html.

[29] Jiyong Jang Dhilung Kirat. 2018. DeepLocker: How AI Can Power a Stealthy New Breed of Malware. Retrieved March 2018 from https://securityintelligence.com/deeplocker-how-ai-can-power-a-stealthy-new-breed-of-malware/.

[30] Marc Ph. Stoecklin Dhilung Kirat, and Jiyong Jang. 2018. DeepLocker: Concealing Targeted Attacks with AI Locksmithing. Retrieved October 2018 from https://i.blackhat.com/us-18/Thu-August-9/us-18-Kirat-DeepLocker-Concealing-Targeted-Attacks-with-AI-Locksmithing.pdf.

[31] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 51–62.

[32] Dennis Distler and Charles Hornat. 2007. Malware analysis: An introduction. *SANS Institute InfoSec Reading Room* (2007), 18–19.

[33] Brendan Dolan-Gavitt and Yacin Nadji. 2010. *See No Evil: Evasions in Honeymonkey Systems.* Technical Report. http://moyix.net/honeymonkey.pdf.

[34] David Reguera Garcia Dreg. 2018. A Tool to Detect and Crash Cuckoo Sandbox. Retrieved October 2018 from https://github.com/David-Reguera-Garcia-Dreg/anticuckoo.
[35] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)* 44 (2012), 6.
[36] Nicolas Falliere. 2007. Windows Anti-Debug Reference. Retrieved October 2018 from http://www.security-focus.com/infocus/1893.
[37] Nicolas Falliere, Liam O. Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White Paper, Symantec Corp., Security Response* 5, 6 (2011), 29.
[38] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. [n.d.]. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 417–426.
[39] Peter Ferrie. 2008. Anti-unpacker tricks, part one. *Virus Bulletin* 4 (2008).
[40] P. Ferrie. 2011. The Ultimate Anti-Debugging Reference. Retrieved October 2018 from http://anti-reversing.com/Downloads/Anti-Reversing/The_Ultimate_Anti-Reversing_Reference.pdf.
[41] Jason Franklin, Mark Luk, Jonathan M. McCune, Arvind Seshadri, Adrian Perrig, and Leendert Van Doorn. 2008. Remote detection of virtual machine monitors with fuzzy benchmarking. *ACM SIGOPS Operating Systems Review* 42, 3 (2008), 83–92.
[42] Shang Gao and Qian Lin. 2012. Debugging classification and anti-debugging strategies. In *4th International Conference on Machine Vision (ICMV 2011): Computer Vision and Image Analysis; Pattern Recognition and Basic Technologies*, Vol. 8350. International Society for Optics and Photonics, 83503C.
[43] Yuxin Gao, Zexin Lu, and Yuqing Luo. 2014. Survey on malware anti-analysis. In *5th International Conference on Intelligent Control and Information Processing (ICICIP '14)*. IEEE, 270–275.
[44] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is not transparency: VMM detection myths and realities. In *HotOS*.
[45] Andrew Go, Christopher del Fierro, Lovely Bruiz, and Xavier Capilitan. 2018. Where We Go, We Don't Need Files: Analysis of Fileless Malware "Rozena". Retrieved October 2018 from https://www.gdatasoftware.com/blog/2018/06/30862-fileless-malware-rozena.
[46] Ian Goldberg, David Wagner, Randi Thomas, Eric A. Brewer, et al. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, Vol. 6. 1–1.
[47] Robert P. Goldberg. 1974. Survey of virtual machine research. *Computer* 7, 6 (1974), 34–45.
[48] GReAT. 2014. The Darkhotel APT. Retrieved October 2018 from https://securelist.com/the-darkhotel-apt/66779/.
[49] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. 2012. Retrieved October 2018 from The Cuckoo Sandbox. https://cuckoosandbox.org.
[50] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. 2008. A study of the packer problem and its solutions. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 98–115.
[51] Thorsten Holz and Frederic Raynal. 2005. Detecting honeypots and other suspicious environments. In *Proceedings from the 6th Annual IEEE SMC Information Assurance Workshop (IAW'05)*. IEEE, 29–36.
[52] Lexi Security Hub. 2014. Overview of the Kronos Banking Malware Rootkit. Retrieved January 2019 from https://www.lexsi.com/securityhub/overview-kronos-banking-malware-rootkit/?lang=en.
[53] Chong Rong Hwa. 2013. Trojan.APT.BaneChant: In-Memory Trojan That Observes for Multiple Mouse Clicks. Retrieved February 2019 from https://www.fireeye.com/blog/threat-research/2013/04/trojan-apt-banechant-in-memory-trojan-that-observes-for-multiple-mouse-clicks.html.
[54] Infosec Institute. 2015. ZEROACCESS Malware - Part 1. Retrieved January 2019 from https://resources.infosecinstitute.com/step-by-step-tutorial-on-reverse-engineering-malware-the-zeroaccessmaxsmiscer-crimeware-rootkit/.
[55] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. 2014. Morpheus: Automatically generating heuristics to detect Android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 216–225.
[56] JOESandbox. 2010. Automated Malware Analysis Report for Wdf01000.sys. Retrieved October 2018 from https://www.joesandbox.com/analysis/45221/0/pdf.
[57] JOESandbox. 2018. shcndhss.exe. Retrieved January 2019 from https://www.joesandbox.com/analysis/50204/0/html.
[58] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. 2009. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*. ACM, 11–22.
[59] Alexandros Kapravelos, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2011. Escape from monkey island: Evading high-interaction honeyclients. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 124–143.

[60] Kaspersky. 2000. VIRUS.WIN32.HIV. Retrieved September 2018 from https://threats.kaspersky.com/en/threat/Virus.Win32.HIV/.
[61] Dhilung Kirat and Giovanni Vigna. 2015. Malgene: Automatic extraction of malware analysis evasion signature. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 769–780.
[62] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2011. Barebox: Efficient malware analysis on bare-metal. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 403–412.
[63] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal analysis-based evasive malware detection. In *USENIX Security Symposium*. 287–301.
[64] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 285–296.
[65] Vitali Kremez. 2017. Let's Learn: Decoding Latest "TrickBot" Loader String Template and New Tor Plugin Server Communication. Retrieved September 2018 from https://www.vkremez.com/2018/07/lets-learn-trickbot-new-tor-plugin.html.
[66] Christopher Kruegel. 2014. How To Build An Effective Malware Analysis Sandbox. Retrieved September 2018 from https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/.
[67] Christopher Kruegel. 2015. Evasive malware exposed and deconstructed. In *RSA Conference*. 12–20.
[68] MacAfee Labs. 2018. Threats Report. Retrieved November 2018 from https://www.mcafee.com/es/resources/reports/rp-quarterly-threats-mar-2018.pdf.
[69] Boris Lau and Vanja Svajcer. 2010. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology* 6, 3 (2010), 181–195.
[70] Kevin Lawton. 2003. Bochs: The open source IA-32 emulation project.
[71] Van Lam Le, Ian Welch, Xiaoying Gao, and Peter Komisarczuk. 2013. Anatomy of drive-by download attack. In *Proceedings of the 11h Australasian Information Security Conference–Volume 138*. Australian Computer Society, Inc., 49–58.
[72] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. 2016. Towards transparent introspection. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*, Vol. 1. IEEE, 248–259.
[73] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 386–395.
[74] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting environment-sensitive malware. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 338–357.
[75] Tao Liu, Nuo Xu, Qi Liu, Yanzhi Wang, and Wujie Wen. 2019. A system-level perspective to understand the vulnerability of deep learning systems. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 506–511.
[76] Leo Loobeek. 2016. Ebowla: Framework for Making Environmental Keyed Payloads. Retrieved October 2018 from https://github.com/Genetic-Malware/Ebowla.
[77] Dejan Lukan. 2014. Pafish (Paranoid Fish). Retrieved September 2018 from https://resources.infosecinstitute.com/pafish-paranoid-fish/.
[78] Malwarebytes. 2018. *SamSam Ransomware: Controlled Distribution for an Elusive Malware*. Technical Report. Malwarebytes Labs. https://blog.malwarebytes.com/threat-analysis/2018/06/samsam-ransomware-controlled-distribution/.
[79] MalwareTech. 2015. Kelihos Analysis, Part 1. Retrieved September 2018 from https://www.malwaretech.com/2015/12/kelihos-analysis-part-1.html.
[80] Steve Mansfield-Devine. 2017. Fileless attacks: Compromising targets without malware. *Network Security* 2017, 4 (2017), 7–11.
[81] Jonathan A. P. Marpaung, Mangal Sain, and Hoon-Jae Lee. 2012. Survey on malware evasion techniques: State of the art and challenges. In *2012 14th International Conference on Advanced Communication Technology (ICACT'12)*. IEEE, 744–749.
[82] McAfee. 2000. The W9x.CIH virus. Retrieved December 2018 from https://home.mcafee.com/virusinfo/virusprofile.aspx?key=10300.
[83] McAfee. 2003. W97M/Opey.bg. Retrieved September 2018 from https://home.mcafee.com/VirusInfo/VirusProfile.aspx?key=100091#none.
[84] McAfee. 2007. W32.Mydoom.M@mm. Retrieved September 2018 from https://www.symantec.com/security-center/writeup/2004-072615-3527-99.
[85] McAfee. 2017. McAfee Labs Threats Report. Retrieved September 2018 from https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-jun-2017.pdf.

[86] Gary McGraw and Greg Morrisett. 2000. Attacking malicious code: A report to the Infosec research council. *IEEE Software* 17, 5 (2000), 33–41.
[87] Microsoft. 2006. Win32/Phatbot.A. Retrieved September 2018 from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=HackTool:Win32/Phatbot.A.
[88] Microsoft. 2017. Worm:Win32/Rbot.ST. Retrieved September 2018 from https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=Worm:Win32/Rbot.ST.
[89] Microsoft. 2018. Acquiring High-Resolution Time Stamps. Retrieved September 2018 from https://docs.microsoft.com/en-us/windows/desktop/sysinfo/acquiring-high-resolution-time-stamps.
[90] Microsoft. 2018. PEB Structure. Retrieved September 2018 from https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx.
[91] Microsoft. 2018. Structured Exception Handling. Retrieved September 2018 from https://docs.microsoft.com/en-us/windows/desktop/Debug/structured-exception-handling.
[92] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 1009–1024.
[93] Carbon Monoxide. 2016. ScyllaHide. Retrieved September 2018 from https://bitbucket.org/NtQuery/scyllahide.
[94] Travis Morrow and Josh Pitts. 2016. Genetic malware: Designing payloads for specific targets. *Talk at Infiltrate* (2016).
[95] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 231–245.
[96] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of static analysis for malware detection. In *23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE, 421–430.
[97] H. Mourad. 2015. Sleeping your way out of the sandbox. *SANS Security Report*.
[98] Microsoft msdn. 2018. Debugging Functions. Retrieved September 2018 from https://msdn.microsoft.com/en-us/library/windows/desktop/ms679303(v=vs.85).aspx.
[99] Microsoft msdn. 2018. ZwSetInformationThread function. Retrieved September 2018 from https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/ntddk/nf-ntddk-zwsetinformationthread.
[100] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. 2009. MAVMM: Lightweight and purpose built VMM for malware analysis. In *Annual Computer Security Applications Conference (ACSAC'09)*. IEEE, 441–450.
[101] Norman. 2018. Norman Sandbox. Retrieved September 2018 from http://www.norman.com.
[102] Kulchytskyi Oleg. 2016. Anti-Debug Protection Techniques: Implementation and Neutralization. Retrieved September 2018 from https://www.codeproject.com/Articles/1090943/Anti-Debug-Protection-Techniques-Implementation-an.
[103] Yoshihiro Oyama. 2018. Trends of anti-analysis operations of malwares observed in API call logs. *Journal of Computer Virology and Hacking Techniques* 14, 1 (2018), 69–85.
[104] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT'09)*, Vol. 41. 86.
[105] David Patten. 2017. The evolution to fileless malware. *Infosec Writers* (2017).
[106] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: In-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the 4th European Workshop on System Security*. ACM, 3.
[107] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A survey of security issues in hardware virtualization. *ACM Computing Surveys (CSUR)* 45, 3 (2013), 40.
[108] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-force: Force-executing binary programs for security applications. In *23rd USENIX Security Symposium (USENIX Security'14)*. 829–844.
[109] Larry Ponemon and Jack Danahy. 2018. *The 2017 State of Endpoint Security Risk Report*. Technical Report. Ponemon Institute. https://www.barkly.com/ponemon-2018-endpoint-security-statistics-trends.
[110] Nguyen Anh Quynh and Kuniyasu Suzaki. 2010. Virt-ice: Next-generation debugger for malware analysis. *Black Hat USA* (2010).
[111] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. 2012. Camouflage in malware: From encryption to metamorphism. *International Journal of Computer Science and Network Security* 12, 8 (2012), 74–83.
[112] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting system emulators. In *International Conference on Information Security*. Springer, 1–18.
[113] Curesec Security Research. 2013. Inkasso Trojaner - Part 3. Retrieved September 2018 from https://curesec.com/blog/article/blog/Inkasso-Trojaner--Part-3-24.html.

[114] Paul Roberts. 2004. Mydoom Sets Speed Records. Retrieved September 2018 from https://www.pcworld.com/article/114461/article.html.
[115] Paul Royal. 2012. Entrapment: Tricking malware with transparent, scalable malware analysis. *Talk at Black Hat* (2012).
[116] Abhishek Singh and Sai Omkar Vashisht. 2014. Turing Test in Reverse: New Sandbox-Evasion Techniques Seek Human Interaction. Retrieved September 2018 from https://www.fireeye.com/blog/threat-research/2014/06/turing-test-in-reverse-new-sandbox-evasion-techniques-seek-human-interaction.html.
[117] Mike Schiffman. 2010. A Brief History of Malware Obfuscation. Retrieved September 2018 from https://blogs.cisco.com/security/a_brief_history_of_malware_obfuscation_part_2_of_2.
[118] Sriranga Seetharamaiah and Carl D. Woodward. 2019. Protecting computer systems used in virtualization environments against fileless malware. US Patent Appl. 15/708,328. Filed date is January 31st., 2019.
[119] Hao Shi and Jelena Mirkovic. 2017. Hiding debuggers from malware with apate. In *Proceedings of the Symposium on Applied Computing*. ACM, 1703–1710.
[120] Tyler Shields. 2010. Anti-debugging—A developers view. *Veracode Inc., USA* (2010).
[121] Michael Sikorski and Andrew Honig. 2012. *Practical Malware Analysis: The Hands-on Guide to Dissecting Malicious Software*. No Starch Press.
[122] Arunpreet Singh. 2017. Malware Evasion Techniques: Same Wolf - Different Clothing. Retrieved October 2018 from https://www.lastline.com/labsblog/malware-evasion-techniques/.
[123] Arunpreet Singh and Clemens Kolbitsch. 2014. Not So Fast My Friend—Using Inverted Timing Attacks to Bypass Dynamic Analysis. Retrieved November 2018 from https://www.lastline.com/labsblog/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic-analysis/.
[124] Sophos. 2015. W32/Agobot-OT. Retrieved October 2018 from https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/W32 Agobot-OT/detailed-analysis.aspx.
[125] Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. LO-PHI: Low-observable physical host instrumentation for malware analysis. In *Annual Network and Distributed System Security Symposium (NDSS'16)*.
[126] Symantec. 2000. Xeram.1664. Retrieved November 2018 from https://www.symantec.com/security-center/writeup/2000-121913-2839-99.
[127] Symantec. 2007. Trojan.Peacomm.C. Retrieved October 2018 from https://www.symantec.com/security-center/writeup/2007-082212-2341-99.
[128] Symantec. 2011. Trojan.Zeroaccess. Retrieved November 2018 from https://www.symantec.com/security-center/writeup/2011-071314-0410-99.
[129] Cylance Threat Guidance Team. 2017. Threat-Spotlight-Satan-RaaS. Retrieved November 2018 from https://threatvector.cylance.com/en_us/home/threat-spotlight-satan-raas.html.
[130] Christopher Thompson, Maria Huntley, and Chad Link. 2010. Virtualization detection: New strategies and their effectiveness. *University of Minnesota* (unpublished).
[131] Joshua Tully. An Anti-Reverse Engineering Guide. Retrieved November 9, 2008 from https://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide.
[132] UIC. 2013. McRat Malware Analysis - Part 1. Retrieved November 2018 from https://quequero.org/2013/04/mcrat-malware-analysis-part1/.
[133] Amit Vasudevan and Ramesh Yerraballi. 2006. Cobra: Fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 15 pp.
[134] Virustotal. 2015. vti-rescan. Retrieved November 2018 from https://www.virustotal.com/en/file/e1988a1876263837ca18b58d69028c3678dc3df51baf1721535df3204481e6a1/analysis/.
[135] Kyle Yang Walter (Tiezhu) Kong. 2013. Unlocking LockScreen. Retrieved November 2018 from https://www.virusbulletin.com/virusbulletin/2013/07/unlocking-lockscreen.
[136] Jeffrey Wilhelm and Tzi-cker Chiueh. 2007. A forced sampled execution approach to kernel rootkit identification. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 219–235.
[137] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy* 5, 2 (2007).
[138] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. 2012. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 189–198.
[139] Rubio Wu. 2017. New EMOTET Hijacks a Windows API, Evades Sandbox and Analysis. Retrieved November 2018 from https://blog.trendmicro.com/trendlabs-security-intelligence/new-emotet-hijacks-windows-api-evades-sandbox-analysis/.
[140] XPN. 2017. Windows Anti-Debug Techniques - OpenProcess Filtering. Retrieved November 2018 from https://blog.xpnsec.com/anti-debug-openprocess/.

[141] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E: Combining hardware virtualization and software emulation for transparent and extensible malware analysis. *ACM SIGPLAN Notices* 47, 7 (2012), 227–238.

[142] Abhishek Singh and Yasir Khalid. 2012. Don't Click the Left Mouse Button: Introducing Trojan UpClicker. Retrieved November 2018 from https://www.fireeye.com/blog/threat-research/2012/12/dont-click-the-left-mouse-button-trojan-upclicker.html.

[143] Mark Vincent Yason. 2007. The Art of Unpacking. Retrieved November 2018 from https://www.blackhat.com/presentations/bh-usa-07/Yason/Whitepaper/bh-usa-07-yason-WP.pdf.

[144] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, et al. 2016. SandPrint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 165–187.

[145] Katsunari Yoshioka, Yoshihiko Hosobuchi, Tatsunori Orii, and Tsutomu Matsumoto. 2011. Your sandbox is blinded: Impact of decoy injection to public malware analysis systems. *Journal of Information Processing* 19 (2011), 153–168.

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

[147] Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang. 2018. Towards transparent debugging. *IEEE Transactions on Dependable and Secure Computing* 15, 2 (2018), 321–335.

[148] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using hardware features for increased debugging transparency. In *2015 IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 55–69.

[149] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. 2013. Spectre: A dependable introspection framework via system management mode. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*. IEEE, 1–12.

Received November 2018; revised June 2019; accepted September 2019