A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion

PC, Mobile, and Web

Alexei Bulazel*
River Loop Security, LLC
alexei@riverloopsecurity.com

Bülent Yener
Department of Computer Science
Rensselaer Polytechnic Institute
yener@cs.rpi.edu

ABSTRACT

Automated dynamic malware analysis systems are important in combating the proliferation of modern malware. Unfortunately, malware can often easily detect and evade these systems. Competition between malware authors and analysis system developers has pushed each to continually evolve their tactics for countering the other.

In this paper we systematically review *i)* "fingerprint"-based evasion techniques against automated dynamic malware analysis systems for PC, mobile, and web, *ii)* evasion detection, *iii)* evasion mitigation, and *iv)* offensive and defensive evasion case studies. We also discuss difficulties in experimental evaluation, highlight future directions in offensive and defensive research, and briefly survey related topics in anti-analysis.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Systems security*; *Virtualization and security*; *Software reverse engineering*;

KEYWORDS

Malware Analysis, Evasive Malware, Dynamic Analysis, Anti-Analysis, Emulation, Virtualization, Anti-Debugging

ACM Reference format:
Alexei Bulazel and Bülent Yener. 2017. A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion. In *Proceedings of Reversing and Offensive-oriented Trends Symposium, Vienna, Austria, November 16–17, 2017 (ROOTS),* 21 pages.
https://doi.org/10.1145/3150376.3150378

1 INTRODUCTION

More than ever, there is a need for automated systems to conduct dynamic malware analysis. Automation is essential in order to keep up with malware developers, who create new malware faster than human analysts can manually analyze it. Automated dynamic analysis systems also perform a valuable role in analyzing unknown software that may or may not be malicious, e.g., for mobile "app stores" vetting submitted apps [126]. These systems observe software as it runs and produce reports on its behavior, removing the burden from time constrained and expensive human analysts. Automated analysis systems generally run software in isolated environments to prevent lasting damage or infection and to enable higher privilege observation.

Unfortunately for defensive researchers, malware can detect analysis by looking for unique systems traits ("fingerprints"), and subsequently behave benignly or exit to evade detection. To counter evasion, researchers have developed techniques to detect and mitigate the behavior in malware.

Our work is timely in reviewing over a decade of research on *i)* evasion techniques for PC, mobile, and web malware analysis systems (and a handful of papers on other platforms), methods for *ii)* detecting evasion, *iii)* mitigating evasion, and *iv)* offensive and defensive evasion case studies. We conclude by critiquing the methodological rigor of work in the field and offering suggestions for directions in future offensive and defensive research. We also briefly survey related topics in anti-analysis and analyzing analysis-resistant malware.

1.1 Overview

1.1.1 Scope. In considering evasive malware, we focus on malware that *evades automated analysis by conditionally executing code* after detecting analysis. Our survey provides more extensive coverage of malware for PC than mobile or web due to the greater number of papers on the topic.

1.1.2 Evaluation. Where statistics are available, we present data from surveyed works on the efficacy of their approaches. Due to extreme variability in sample sizes and evaluation methodologies, as well as challenges in establishing ground truth about evasion in wild malware, it is difficult to effectively compare various analysis approaches. We do not empirically compare approaches in their efficacy, but offer tables summarizing surveyed works, and inline discussion with each section.

1.1.3 Organization. In the first two-thirds of this paper we offer a comprehensive survey of prior research on evasive malware, covering analysis system fingerprinting (§2), evasion detection (§3), evasion mitigation (§4), and case studies on evasion on the wild (§5). In the final third we present discussion, critiquing scientific

*Work done in part while author was with Department of Computer Science, Rensselaer Polytechnic Institute.

ROOTS, November 16–17, 2017, Vienna, Austria
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5321-2/17/11…$15.00
https://doi.org/10.1145/3150376.3150378

rigor in the field, proposing future research topics (§6), and surveying related areas of study (§7). We organize sections by broad approaches to these problems, while also providing an exhaustive survey of works.

1.2 Platforms

In this section, we briefly discuss the design of surveyed automated dynamic malware analysis systems. In these systems, malware execution is monitored using various types of instrumentation, generally either in-system (e.g., an observing process, injected libraries, function hooks, or kernel drivers), or out-of-system (e.g., virtual machine introspection, hypervisor traps, network monitoring, or hardware instrumentation).[1]

Transparency: We use the term "transparency" to refer to the property of making analysis systems indistinguishable from non-analysis systems, even to a dedicated adversary [65].[2] We note Dinaburg et al.'s [46] five requirements for transparency: *i)* higher privilege (an analysis system must run at higher privilege than the malware under analysis), *ii)* no non-privileged side effects (side effects from the analysis system's presence must occur at a higher privilege level than the malware's, *iii)* identical instruction execution semantics, *iv)* transparent exception handling (no difference in exception handling as a result of analysis), and *v)* identical measurement of time.

1.2.1 *Virtualized Analysis Systems.*

We review virtualization approaches, as many surveyed automated analysis systems have been built on virtualized infrastructure. The term "guest" refers to the system being virtualized, while "host" refers to the system doing the virtualization. Our background is high level, and our statements cover virtualization approaches for analysis *in general.* The term "virtual machine monitor" (VMM) (or simply "virtual machine" or VM) is used to refer to the code responsible for virtualizing the guest system. In table 1, we compare various approaches to automated malware analysis proposed in surveyed literature.

Emulators: Emulators run all guest hardware instructions in software. Because these instructions are run in software, there is no requirement that the virtualized guest system be of the same architecture as the host. Emulation gives researchers deep insight into malware behavior, as they can easily control, monitor, and modify system state. However, the increased introspection and privilege afforded by emulators generally comes at the cost of reduced transparency. The QEMU [20] emulator has been used in many surveyed works.

Hosted Virtual Machines: We use Pék et al.'s [130] term "hosted" to refer to virtual machines which run within existing host operating systems. These VMs are distinguished from emulators by their requirement that virtualized guest systems be of the same architecture as the host, as they run the majority of instructions from the guest system directly on hardware. Privileged sensitive instructions may dealt with using a variety of techniques, including paravirtualization, binary translation, or hardware-assisted virtualization. Popular examples of these types of VMs include VMware [3] and VirtualBox [2]. While they have not been used in many *surveyed* works on automated malware analysis, these systems have been used in many research projects not covered here, and are extensively used in industry by malware researchers for manual (i.e., debugging) and automated (e.g., Cuckoo Sandbox [1]) dynamic analysis.

Hypervisors: Hypervisors (a.k.a. "native virtual machines," "bare metal virtual machines," or "type 1 hypervisors") perform virtualization without a host operating system between them and physical system hardware. These systems run directly on physical hardware, and virtualize code of the same architecture as the host. Due to their relatively lower overhead, hypervisors are generally more transparent than other virtualization systems. The Xen hypervisor [15] has been used in several surveyed analysis systems.

1.2.2 *Bare Metal.*

Recent research has used "bare metal" analysis to counter virtualization-resistant malware. In bare metal analysis systems, researchers do not virtualize their analysis environment, instead they run and analyze malware on real hardware. Bare metal analysis systems may be more transparent than virtualized analysis systems, but this transparency comes as a trade-off for less insight into malware behavior and challenges in scalability [94, 96, 121, 147, 163].

1.2.3 *Design of Android Analysis Systems.*

While Android is based on Linux, the operating system has a distinct API with a unique Java bytecode VM and native code-based application architecture. Surveyed Android analysis systems have generally been implemented using emulators (e.g., QEMU), though recent literature has described bare metal Android analysis.[3]

1.2.4 *Design of Web Browser Analysis Systems.*

Web malware[4] analysis systems generally take one of two forms: instrumented browsers (a.k.a. "high-interaction honeyclients"), or browser emulators (a.k.a. "low-interaction honeyclients"). Browser analysis systems may use various approaches to discover web malware: waiting for submissions [40], searching for content with web crawlers [120, 174], or analyzing browser extensions rather than web pages [89, 97].[5]

In instrumented systems, web browsers are run and instrumented with in or out-of-system analysis tools, which look for actions indicative of web malware during browser execution (e.g., new processes or files created) [40, 120, 174]. While these systems are usually run inside VMs, bare metal browser analysis systems have also been proposed [174].

Browser emulators are not built on real browsers, instead they emulate them with custom HTTP, HTML, JavaScript, etc., engines instrumented to detect malicious behavior [40, 122].[6] These systems

[1]Readers interested in more background on on automated dynamic malware analysis systems are referred to Egele et al.'s [52] survey of the field at large circa 2012. Readers interested in learning more about out-of-system virtual machine analysis are referred to Bauman et al.'s [17] survey and Jain et al.'s SoK [77].

[2]We paraphrase Garfinkel et al.'s [65] definition of transparent VMMs: "VMMs that are indistinguishable from native hardware, even to a dedicated adversary."

[3]We refer readers interested in learning more about Android automated dynamic malware analysis systems to Neuner et al.'s survey [123].

[4]Note that many web browser analysis systems are more concerned with web pages that deliver browser exploits and unwanted content, though we use the term "malware" as is standard in surveyed literature.

[5]We refer readers to Rajab et al.'s [138] paper for additional discussion of automated web malware analysis approaches and evasive web malware trends.

[6]These systems implement the functionality of a "real" browser (e.g., Firefox, Chrome, Internet Explorer), but are purpose-built custom applications for malware analysis.

Table 1: In this table, we compare analysis system architecture approaches, and their relative pros and cons for analyzing evasive malware. These comparisons are general high-level statements relative between the classes of systems.

	Emulator	**Hosted VM**	**Hypervisor**	**Bare Metal**
Architecture	All code emulated in software	Majority of non-privileged code run on hardware	Virtualization directly on hardware	No virtualization
Transparency	Low	Low	Medium	High
Common Detection Techniques	CPU and timing attacks, unique emulated hardware	Hosted VMs suffer from the same issues as emulators though may be marginally more transparent	CPU behavior edge cases, low but non-zero timing overhead	May be detected on a case-by-case basis - see discussion in §4.7-4.8 & §6.2.1
Advantages	Introspection, total control over state	Ease of use, introspection, control over state	Minimal overhead, close to hardware	No virtualization fingerprints
Disadvantages	Susceptibility to CPU semantics and timing attacks	Designed for compatibility over transparency [65]	Less introspection ability, still detectable	Difficult to scale, expensive, loss of introspection
Examples	QEMU [20], Bochs [105]	VMware [3], VirtualBox [2]	Xen [15], VMware ESXi [3]	See §4.7

generally look for known malicious signatures (e.g., a JavaScript function call with parameters known to trigger a particular vulnerability) during analysis.

2 DETECTING ANALYSIS SYSTEMS

In this section, we review analysis system traits, or "fingerprints," that malware may use to detect analysis (enabling it to subsequently behave benignly in order to evade detection). In surveying fingerprinting techniques, we mainly draw upon offensive research on fingerprinting techniques and defensive work analyzing evasion techniques observed in wild malware. Table 2 summarizes fingerprints discussed in this section.

Some fingerprints may not fit cleanly into one category, and instead may be classified in two or more.[7] We note that analysis system fingerprints may come from a variety of sources including underlying technologies facilitating analysis,[8] system configuration,[9] or analysis systems themselves.[10]

2.1 Environmental Artifacts

Environmental artifacts are among the simplest analysis system fingerprints. These fingerprints are uniquely distinguishing traits of the analysis system's environment, such as usernames, files, system settings, running processes, drivers, mutexes, etc.

Environmental fingerprints may stem from unique system settings, analysis system instrumentation (e.g., drivers or files on disk), or underlying infrastructure (systems which virtualize unique or legacy hardware, or unique resource allocations, e.g., single CPU systems in the current year) [24, 36, 40, 73, 88, 104, 111, 113, 137, 140].

For example, Lindorfer et al. [111] present examples of environmental artifacts they observed malware using to detect the Anubis system, such as the username `USER`, the presence of a file at `C:\\exec\exec.exe`, or a process running on the system called `popupkiller.exe`.

Chen et al.'s [36] 2008 taxonomy of evasion techniques in malware discusses a number of environmental artifacts used by evasive malware in the wild.

2.2 Timing Attacks

Timing is incredibly challenging to accurately simulate on virtualized hardware, so inconsistencies in timing have frequently been used to detect virtualization [65]. Sophisticated attacks involving unique timing inconsistencies are possible [26, 36, 60, 65, 71, 73, 130, 137, 160, 169], though malware in the wild often uses simple attacks, just checking a threshold for how long a set of operations should take (e.g., invoking library functions, making system calls, or executing single instructions) [36, 111]. Ho et al. [71] point out that virtualization can be detected with timing attacks from JavaScript running within a web browser.

Creating timing accurate VMMs is a difficult problem, as efforts to hide timing discrepancies may in fact exacerbate the issue rather than fix it. Garfinkel et al. [65] note that the hardware translation lookaside buffer (TLB) is shared between hosts and VMMs, and thus may be detectably strained by VMM overhead. Efforts to create a software-simulated "virtual TLB" for VMM use can incur performance overhead, leaving the VMM more open to other timing-based detection attacks. The authors also point out the intractability of preventing time measurement through *remote* channels as a "VM can use nearly any communication from the outside world as a clock." Attempting to hide timing skews though "time dilation" (i.e., making virtual time progress more slowly than real time) poses "extreme engineering hardship and huge runtime overhead," and may make VMMs more vulnerable to detection as it increases the distance between local and remote time data.

Timing attacks are not limited to virtualization, though unlikely, bare metal analysis systems may be detected through timing overhead introduced by hardware instrumentation [156].

2.3 CPU Virtualization Artifacts

Commonly known as "red pills," instructions which behave differently when run on virtualized CPUs can be used to detect virtualization. Examples include instructions which set status flags

[7] e.g., evidence of web browsing history is an environmental artifact but is also evidence of user presence; a unique cycle count for given CPU instruction is both a CPU red pill and a timing fingerprint

[8] e.g., CPU or timing discrepancies from VMMs and virtualized hardware

[9] i.e., traits of the system which can be used to identify it, but are not actually related to analysis, e.g., usernames, system IP addresses, evidence of use

[10] e.g., installed drivers, timing or performance overhead introduced by analysis, unique user input patterns

Table 2: In this table we summarize our categories of analysis system fingerprints and present examples of each.

Type	Definition	Examples
Environmental Artifacts [24, 36, 40, 65, 73, 88, 104, 111, 113, 137, 140]	Unique distinguishing values from an analysis system's environment arising from virtualization infrastructure, analysis instrumentation, or system configuration	Unique usernames, system settings, files present on disk, installed drivers
Timing [26, 36, 60, 65, 71, 73, 111, 130, 137, 160, 169]	Unique timing skews, discrepancies, and overhead	Timing overhead resulting from virtualization or analysis instrumentation, unique skews introduced by attempts to hide this overhead
CPU Virtualization [8, 56, 57, 104, 114, 115, 128, 137, 148, 178]	Discrepancies in CPU instruction semantics and interpretation	Incorrectly emulated instructions, instructions erroneously accepted as valid, incorrectly implemented model-specific behavior
Process Introspection [21, 125, 134, 159, 178]	Runtime program state artifacts, injected code or libraries, hooks	Injected libraries, function hooks, memory or register state
Reverse Turing Tests [21, 45, 49, 49, 88, 113, 116, 123, 133, 138, 140, 174, 186]	Evidence of a human user on the system	Lack of keyboard or mouse input, copy-paste clipboard data, web browser history, login times
Network Artifacts [8, 21, 49, 67, 156]	Discrepancies in network behavior or access	Failure to handle certain protocols, network isolation, fingerprintable values in emulated protocols
Mobile Sensors [24, 45, 81, 113, 121, 126, 134, 140, 169]	Artifacts resulting from emulation (or lack thereof) of mobile device sensors	Hardcoded return values, unsupported common sensors, failure to model feedback between sensors
Browser Specific [31, 40, 88, 138, 174]	Fingerprints unique to web browser analysis systems	Scripting interpretation bugs or discrepancies, HTML parsing discrepancies, incomplete DOM implementation, system-specific canvas noise

incorrectly, erroneously alter memory, do not generate exceptions correctly, or are incorrectly accepted or rejected by a CPU emulator [8, 56, 57, 65, 104, 114, 115, 128, 137, 148, 178]. We also note the possibility of attacks not directly related to inconsistencies in CPU *semantics*, such as discrepancies in values returned by the `SIDT` or `SGDT` instructions in virtualized environments [65].

We refer readers interested in learning more about CPU red pills to Paleari et al.'s [128] seminal work on large-scale red pill discovery.

2.4 Process Introspection Artifacts

Process introspection artifacts are fingerprints associated with the runtime state of a program that the program may detect by looking into its own state (i.e., memory or registers). Examples include injected libraries, function hooks, dynamic binary instrumentation, or other in-process observables [21, 88, 125, 159]. Eccentricities in instruction and memory handling can also be used to detect virtualization (e.g., the inability to properly handle self-modifying code) [134, 178].

2.5 Reverse Turing Tests

As automated analysis systems analyze malware without human interaction, some malware may choose to run only if it detects a human using the system (or finds evidence that a human *has* used the system). These "reverse Turing Tests"[11] may be grouped into three broad categories: *1)* passive observation of live human computer use (e.g., mouse movement, typing cadence, process creation), *2)* active interaction with a human user (e.g., making the user click a dialog box created by the malware before it runs), and *3)* passive wear-and-tear artifacts.

[11] An inversion of the traditional Turing Test where a *computer* attempts to distinguish if it is interacting with a human or a computer

Malware may look for traits such as mouse or keyboard inputs, meaningful data on the copy-paste clipboard, file metadata (e.g., sizes, types, access times, "recently opened" lists in applications), login times, installed software (including plugins in web browsers), web history, or camera roll photos (in mobile analysis systems) [21, 49, 91, 113, 116, 133, 140, 186]. Many Android analysis systems simulate user interaction, but these interactions may be fingerprintable through timing or interaction patterns [45, 123, 140]. Malicious websites may only act after detecting user interaction such as mouse movement, clicking, or scrolling [49, 88, 138, 174].

Researchers have examined the challenges of analyzing malware driven by user interactions (e.g., Android apps that only exhibit malicious behavior on certain Activities). We consider this behavior to be outside the scope of our survey, though we briefly discuss the topic in §7.5.

Yokoyama et al.'s [186] work on SandPrint has highlighted the potential for detecting systems which exhibit traits indicative of automated analysis, but that are unlikely to be observed in systems with real human users. The authors cite features related to human interaction such as the presence of data in the system clipboard, file access times, and system and login session uptimes.

As Miramirkhani et al. [116] discuss, subtle "wear-and-tear" artifacts indicating human use of computers (e.g., web history, files left in the system recycling bin, the presence of crash dumps, etc.) can be used to detect analysis systems. These artifacts may be useful to malware authors, as they do not rely on just detecting virtualization, and they do not require a human user to be present at the time of malware infection (as detection techniques such as waiting for mouse movement or clicking do).

2.6 Network Artifacts

Unique network behavior and traits in analysis systems can be used for analysis detection. Examples include fixed IP addresses

[187], extremely fast internet service [49], network simulation [21], or even network isolation [156]. Some emulated systems may be detected by their failure to handle certain types of network communications, such as the default Android SDK emulator's inability to forward ICMP packets [112, 113, 140].

At the intersection of environmental, user presence, and network artifacts, Miramirkhani et al. [116] point out that evidence of network activity (or lack thereof) may itself be an indicator of automated analysis. The authors show that artifacts of network activity such as certificate revocation lists, DNS cache, and cached Wi-Fi SSIDs can indicate that a real human has previously used a system.

2.7 Mobile Sensors

Researchers implementing virtualized analysis systems for mobile devices face unique challenges in emulating the myriad of sensors built into these devices, including multiple radios (e.g., cellular, Wi-Fi, Bluetooth, NFC), cameras, microphones, GPS, accelerometers, gyroscopes, thermometers, barometers, proximity sensors, light sensors, magnetic sensors, humidity sensors, air pressure sensors, geomagnetic field sensors, and voltmeters, among others [24, 45, 81, 113, 121, 126, 134, 140, 169]. Sensor readings cannot be trivially emulated because they can be easily affected by apps and may have complex interactions [121, 169].[12] Boomgaarden and Corney's thesis gives extensive treatment to sensor-based Android evasion, and includes over 200 pages of code demonstrating attacks [25].

While these attacks may be adapted to sensors built into traditional PCs, or accessible from web applications, we are not aware of any research or malware in the wild using them.

2.8 Browser Specific Fingerprints

Web browser-based systems are vulnerable to their own unique types of fingerprinting. We draw upon and extend Rajab et al.'s [138] and Kapravelos et al.'s [88] taxonomies of browser fingerprints.

2.8.1 Scripting Environment Compatibility. Inconsistencies in web scripting environments between various browsers and browser emulators may be used by malware for fingerprinting.

JavaScript: JavaScript implementation is not consistent between various browsers and engines, so unique discrepancies in language parsing, exception handling, scoping, and other behaviors may be used to fingerprint specific JavaScript engines. Engine discrepancies pose a particular danger to browser emulators which *identify as* a certain browser, but do not faithfully emulate its engine's quirks [40, 90, 138]. Differences in JavaScript handling within PDFs has also been used to evade analysis systems [90].

ActiveX: As with JavaScript, unique ActiveX emulation inconsistencies may be used to fingerprint browser emulators that support it [40, 90].

VBScript: Kapravelos et al. [90] describe an attack observed in the wild where VBScript (rather than JavaScript) was used to launch an ActiveX exploit; presumably to evade analysis, as most browser emulators do not support the language.

[12]e.g., device vibration can be verified with an accelerometer, CPU strain can increase internal temperature or deplete battery, audio output can be picked up by microphones

2.8.2 Parser Compatibility. Unique differences in HTML parsing between browser engines can be used to trigger JavaScript code execution allowing for fingerprinting [138].

2.8.3 DOM Completeness. Browser emulators may feature incomplete Document Object Model (DOM) emulation which can be detected [138].

2.8.4 HTML5 Canvases. Bursztein et al.'s [31] work on mobile device fingerprinting proposes differentiating between bare metal and emulated systems by using unique software and hardware "noise" produced during HTML5 canvas rendering.

2.8.5 Shellcode-Time Detection. Kapravelos et al. [88] note the possibility of web-based exploits conducting analysis detection during shellcode execution rather than through browser scripting engines.[13]

2.8.6 Referrer Checks. Malicious web sites may choose to only deliver malware when contacted with specific or non-empty HTTP referrers in request headers [40, 49, 88]. For example, attackers may compromise a specific site and place a link to malicious content on it, but only serve the content when referred from the compromised site. This style of evasion happens before the malware is actually delivered, but note it here due to the unique considerations of web-based malware.

2.9 Discussion

In this section, we have surveyed fingerprinting techniques that malware may use to detect analysis.

We note that many fingerprinting techniques are related to virtualization, as analysis systems, both dynamic and manual, are often virtualized. As we discuss in §4.7, a number of researchers building automated analysis systems have proposed bare metal analysis systems in response to the challenges to transparency posed by virtualization.

Across all types of automated analysis systems, we note that user-related fingerprints are likely the most difficult to mitigate. By their very nature of being *automated*, surveyed analysis systems do not have human operators, and are thus likely to remain vulnerable to these attacks despite efforts to increase analysis transparency. As researchers have shown, efforts to spoof human interaction may themselves be fingerprintable.[14] Fingerprinting techniques range from being simple to use and mitigate against (e.g., waiting for clicking), to extremely challenging to use but also difficult to mitigate (e.g., heuristic models of human behavior or statistical modeling of file system metadata distribution).

Most surveyed works have used human-driven fingerprint discovery (or observation in wild malware) to find fingerprints. Future researchers may look to work on automating fingerprint discovery, such as Jing et al.'s [81] Morpheus, a tool for *automatically* finding fingerprints in QEMU and VirtualBox-based Android emulators.

[13]Evaluating 9,230 malicious URLs in 2011, the researchers did not find shellcode-based attacks in the wild, though they demonstrated the possibility.
[14]See §6.3.4

3 DETECTING EVASION

Detecting evasive malware is an important concern for developers of automated analysis systems. In this section we survey papers on detecting, but not mitigating, evasive malware behavior, though we make an exception by including Kang et al.'s [86] work due its early position in the field and relevance to subsequent research.

3.1 Detecting Known Anti-Analysis Techniques

In cases where anti-analysis techniques are known in advance, defensive researchers have proposed looking for these behaviors in execution traces.

In an early paper in the field, Lau et al. [104] proposed DSD-Tracer, a framework that combines static and dynamic analysis to detect malware employing known VMM evasion techniques. DSD-Tracer runs malware in Bochs[15] and VMware and obtains instruction execution traces from both systems. Python scripts run over the execution traces analyze malware behavior at each captured CPU tick, looking for known VMM detection techniques, such as searches for environmental artifacts or the execution of CPU red pills. Evaluating DSD-Tracer on a set of "around 400" malware samples packed by 193 packers, the authors found that 2.13% exhibited VMM evasion.

Similarly, Brengel et al. [26] propose looking for "excessive" use of "suspicious instructions" (e.g., `rdtsc` and `cpuid`) as a defense against timing attacks that they demonstrated for Intel VT-x based hypervisors.

3.2 Multi-System Execution

A number of researchers have used multi-system execution to detect evasion. In these research efforts, malware is run on two or more analysis systems, and information about execution is compared in order to determine if the malware exhibits evasion in one or more of the systems.

3.2.1 Detection At The Instruction Level. At the lowest level, behavioral divergence may be detected through differences in instruction traces captured from executing a malware sample in different environments. Kang et al.'s [86] work on mitigating evasion proposes a method for detecting a point of divergence in a malware execution instruction trace. After capturing instruction traces from Ether [46][16] and Temu [185],[17] the authors' algorithm looks for the first instruction found in the Temu trace, but not the Ether trace. The algorithm then constructs an alignment in the two traces between the beginning of execution and the divergent instruction in order to track down the root cause of divergence, which is subsequently extracted as an instruction slice. Extraneous factors such as time of execution, miscellaneous hardware differences, or the timing of user inputs used to start the malware may cause irrelevant differences in execution traces. The authors account for these differences by tracing divergence to the first instruction seen in one trace but not the other (unique by program counter location and instruction bytes). We discuss Kang et al.'s work further in §4.4.

[15]An x86 emulator [105]
[16]A Xen-based analysis platform, discussed in §4.6
[17]A QEMU-based analysis system

3.2.2 Detection At The System Call Level. Balzarotti et al. [12] propose comparing system calls made in different environments as a means of detecting divergence. The authors criticize Kang et al.'s [86] work, noting that with instruction trace-based divergence detection, analysis systems are prone to detect deviations that are spurious and not related to anti-analysis (even in light of attempts to minimize these differences). Implemented as an extension to Anubis [19],[18] Balzarotti et al.'s research takes a system call trace of malware execution on a reference system using a kernel driver, and then replays the results of these calls on the system under test when the same input values are given. This allows the test system to appear closer to the reference system, and ensures that analysis detection is not based on trivial environmental attacks. After obtaining a system call trace in the test system, the researchers were able to compare the two traces to discover divergence, and account for the fact that no two executions of the same program are often identical at the instruction level.

To test their work, Balzarotti et al. collected ten malware samples from the Anubis database known to evade QEMU, and ran them under both QEMU and VMware. The system was able to identify detection attempts in all ten samples.

Kirat and Vigna's [95] MalGene locates evasive behavior using a system call trace alignment technique inspired by bioinformatics research in DNA and protein alignment. In testing, the authors collected system call traces from a set of 2,810 evasive malware samples (as identified by BareCloud [96][19]) executed in Anubis and Ether [46]. MalGene was able to automatically extract analysis "evasion signatures"[20] from the samples, and group them by the use of 78 similar evasion techniques.

3.2.3 Detection Through Persistent Changes To System State. Lindorfer et al. [111] propose Disarm, a tool for detecting malware that exhibits semantically different behavior when running in different analysis systems. Disarm runs malware multiple times in different analysis environments (Anubis, and three custom QEMU-based systems with in-system analysis drivers and various unique system settings) and captures persistent changes to system state (e.g., file system or registry writes, starting or stopping of processes and services, or network activity) made during each run. The system works with both in and out-of-guest monitoring systems for capturing execution information, and is able to detect evasive behavior based on malware sensitivity to either of these types of instrumentation. Running malware under test multiple times within each environment allows the authors to normalize captured behavior profiles, removing misleading and irrelevant differences in observed behavior. Only semantically divergent behaviors are considered different; for example, two different systems may have different system languages, causing the names of Windows directories to vary,[21] but a write to the system `Program Files` directory is semantically the same regardless of the literal path to the directory.

After capturing and normalizing behavioral profiles from two analysis environments, the profiles are analyzed by calculating their

[18]A QEMU-based analysis system
[19]An analysis system discussed in §3.2.4
[20]"a set of system call events, user API call events, and comparison events that are used as the basis for evading the analysis system"
[21]e.g., `C:\\Program Files` on an English language system versus `C:\\Arquivos de Programa` on a Spanish language system

Table 3: In this table we compare of evasion detection approaches. We note a general evolution towards increasingly abstracted high-level monitoring, and analyses drawing upon increasingly complex mathematical and statistical methods.

Tool	Analysis Input	Detection Method	Year
DSD-Tracer [104]	CPU instruction trace	Prior known anti-analysis techniques	2008
Kang et al. [86]	CPU instruction trace	Instruction-level divergence	2009
Balzarotti et al. [12]	System call trace	System call-level divergence	2010
Disarm [111]	Persistent changes to system state	Jaccard distance over normalized behavioral profiles	2011
Revolver [90]	Web script ASTs	AST comparison against known malicious scripts	2013
BareCloud [96]	Transient behavioral profile	Hierarchical similarity of transient behavioral profiles	2014
MalGene [95]	System call trace	Bioinformatics-inspired sequence alignment algorithm	2015

Jaccard distance, and comparing the final score to a threshold value for sufficiently divergent behavior.

Using Disarm, the authors analyzed 1,686 samples submitted to Anubis in the four analysis systems and found that 58.72% of samples behaved differently in at least one of them. The authors also discovered several previously unknown fingerprints used by malware to detect Anubis.

3.2.4 Profiling Malware Behavior. Kirat et al. [96] proposed an evasion detection system using bare metal analysis with their work on BareCloud. The system is unique in its comparison methodology for execution traces; similar to Disarm [111], rather than simply considering *how* malware behaved (e.g., tracing instructions or system calls), BareCloud analyzes "resultant behavior profiles" - *what* malware did (e.g., modified a file in the `Program Files` directory, or made a connection to a command and control server).

BareCloud captures data about execution for comparison and divergent-behavior detection from four analysis systems: a hypervisor (Ether [46]), a hosted virtual machine (Cuckoo Sandbox), an emulator (Anubis), and the BareCloud bare metal analysis system. In order to minimize spurious behavioral differences, the authors loaded each analysis system with identical software, and took measures to minimize differences in external environmental factors. Before comparing behavior profiles, the profiles are processed to remove non-malware related system changes and to normalize semantically similar actions. Taking inspiration from "hierarchical similarity" as used in text similarity and computer vision research, the authors construct behavioral similarity hierarchies out of information about malware actions, considering factors such as object names, types, and operation attributes.

Testing a ground truth set of 234 evasive and non-evasive malware samples (established through manual human reverse engineering), the authors demonstrated that their algorithm achieved 100% detection precision, while Jaccard distance-based similarity calculations on the same data could only achieve 88.54% detection precision at max. In additional experimentation with 110,005 malware samples submitted to Anubis over a four month period during summer 2013, BareCloud detected evasive behavior in 5,835 of the samples.

3.3 Web Code Similarity Detection

In a rare paper focused on detecting evasive *web* malware, Kapravelos et al. [90] propose Revolver, a system for identifying analysis evasion in JavaScript web malware. Revolver begins by collecting sets of "benign" and "malicious" web pages classified by an existing web malware analysis system. The abstract syntax trees (ASTs) of individual scripts in each page are then extracted and compared in pairs. Similar pairings of malicious and benign ASTs are given further inspection, as the "benign" AST may be in fact be an evolution of the malicious code with evasive behavior added (hence the classification as benign by analysis systems). If the only difference between the ASTs is the insertion of additional control flow nodes, the benign-classified AST is assumed to be evasive, as the additional nodes are likely evasion related. Revolver's AST-based comparisons may also be used to find other variations in web malware, such as evolution or insertion of malicious code into benign scripts.

In evaluation, the authors analyzed web pages submitted to Wepawet [40],[22] collecting 6,468,623 web pages submitted to the service, 265,692 of which were classified as malicious. Analyzing 20,732,766 benign scripts and 186,032 malicious scripts from the pages, 4,147 were found to exhibit evasive behavior using 155 unique evasion techniques.

3.4 Discussion

In this section, we have surveyed approaches to detecting evasive malware.

Other than DSD-Tracer and Revolver, all of the surveyed works in this section have relied on multi-system execution. While multi-system execution may be an effective method for dealing with malware that uses potentially unknown evasion techniques, it is a resource intensive process fraught with challenges. Multi-system execution may be hindered by malware that can evade analysis across multiple executions, e.g., malware that can detect and evade multiple types of virtualization, or malware that evades analysis by using user interaction-based fingerprints. As shown in table 3, multi-system execution approaches have evolved over time to use increasingly complex algorithmic analyses while also working over more highly abstracted execution traces.

Relative to mitigation research, few works have looked *solely* at evasion detection. We posit that the imbalance of research on evasion detection versus mitigation is due to the fact that systems that can *detect* evasion have already solved much of the issue of evasion, and are most useful if they follow through by mitigating it as well. Detection alone does not take on the main challenge of evasive malware, the fact that it prevents effective analysis, whereas mitigations *enable* analysis.

[22] A web malware analysis system based on jsand [40]

4 MITIGATING EVASION

In response to the proliferation of evasive malware, researchers have proposed *mitigations* to counter malware's evasion. While some mitigation techniques attempt to mitigate specific types of evasion when detected, others aim to more broadly increase transparency and do not need to detect attempted evasion. Evaluating mitigation techniques can be challenging, as researchers must define their own metrics for the efficacy of proposed mitigations (often, that malware exhibited more or previously unobserved activity relative to other executions). Researchers must also minimize or eliminate discrepancies between analysis sessions that could cause divergent behavior for reasons unrelated to evasion (e.g., different time of execution or discrepancies in environment).

4.1 Binary Modification

Though not fully *automated*, Vasudevan et al.'s [167] COBRA was one of the first malware analysis systems specifically designed for transparency [12].[23] Taking inspiration from the design of QEMU, COBRA emulates codes in blocks, dynamically removing or rewriting instructions to prevent detection. In order to prevent malware from detecting modification, COBRA only modifies instructions in its execution blocks, not the malware's actual code resident in memory. The authors note that COBRA's code modification could be detected by checking the address of the program counter and stack state during execution. In evaluation, the authors used COBRA to automate analysis of `W32/Ratos`, an evasive trojan.

4.2 Hiding Environmental Artifacts

In early work on automated malware analysis, Willems et al. [177] proposed CWSANDBOX, a VMware-based system that uses in-system instrumentation.[24] The system uses "rootkit-like" functionality to hide its environmental artifacts (e.g., files, registry entries, mutexes) from malware by hooking operating system functions that could reveal their presence. In evaluation, the authors used CWSANDBOX to analyze 6,148 malware samples from 2006, but did not evaluate its efficacy against evasive malware. Oberheide [125] later pointed out several ways in which CWSANDBOX may be detected and evaded.

4.3 Path Exploration

Path exploration-based approaches force malware code execution down multiple possible conditional branches in order to achieve better code coverage and to mitigate evasion. These techniques have wide applicability in malware analysis beyond countering evasion (e.g., exploring malware behavior when command and control infrastructure is not available, or analyzing trigger-based malware).

Note that path exploration techniques may be fragile and prone to driving execution down paths in which program state is corrupted, leading to crashes. Malware can actively subvert exploration efforts with techniques drawn from anti-symbolic execution obfuscation [13] or anti-fuzzing [175].

[23]COBRA is one part of the WILDCAT analysis framework [164], alongside SAKTHI [165] (binary instrumentation through function hooking), VAMPIRE [166] (stealth breakpoints) and SPIKE (stealth breakpoint based hooking) [168].

[24]Including a DLL *injected directly into* malware under analysis, easily detected through process introspection [125]

4.3.1 PC. Moser et al. [118] propose forcing exploration of multiple execution paths in order to identify conditionally-executed malware behavior. Though the authors' focus is not specifically on mitigating evasive behavior, but more broadly on exploring code paths and analyzing trigger-based malware, we note it here due to its early and seminal influence on the field. The system runs malware under test in QEMU, and logs system calls made during execution. By tainting inputs given to the program, the system is able to track when input values are used to make control flow decisions. The program can then be reanalyzed using different inputs so as to cause exploration of alternative paths of execution. In evaluation of 308 malware samples from 92 families, the authors found that 229 acted conditionally based on at least one tainted input source, and 172 (55.8%) exhibited additional behavior under forced path exploration.

While more focused on analyzing *trigger-based* malware, we also note Brumley et al.'s [29] seminal work on MINESWEEPER.[25] Similar to Moser et al.'s [118] work, but able to handle more complicated formulas and bit-level operations; the system can automatically detect trigger-based behavior, find conditions to trigger it, and identify inputs satisfying those conditions. MINESWEEPER represents inputs to potential triggers (e.g., timing, network input, or keyboard input) symbolically and executes them code handling them symbolically, while other inputs and code are handled concretely. When trigger inputs are used in conditional branching, MINESWEEPER can solve for input values that trigger execution down each path, and can then force code execution down each in order to enable observation of the code's behavior. In evaluation, the authors tested four malware samples and identified multiple conditional branches based on trigger inputs in each.

4.3.2 Web. Cova et al.'s [40] JSAND forces the invocation of JavaScript functions in web malware if the functions were not called during regular execution. This exploration serves as a mitigation against potential evasion attacks, as well as a method of detecting targeted malware (e.g. malware which only runs if a browser's language is set to a certain value).

Kolbitsch et al.'s ROZZLE [102] seeks to identify web malware by using multi-path JavaScript exploration. ROZZLE's focus is on analyzing environmentally sensitive web malware targeting specific browser configurations, through the authors note the applicability of the approach to mitigating evasion.

Kim et al.'s J-FORCE [92] forces path exploration by recording branch outcomes and mutating them. As forced execution systems may crash because of references to missing DOM elements, J-FORCE is able to dynamically create new DOM elements on demand as needed to allow execution to continue. In evaluation, the authors uncovered hidden code behind evasion checks in 41 of 50 Exploit Kit exploits, and discovered ad injection in 322 of 12,132 Chrome browser extensions.

4.3.3 Other Works. Peng et al.'s [132] X-FORCE forces malware execution without using any inputs or environmental setup. Johnson et al.'s [83] work forces execution with instrumented Dalvik bytecode in order to discover potential behaviors in Android apps,

[25]MINESWEEPER is a plugin for TEMU, a dynamic analysis platform from the BitBlaze binary analysis program [155].

Table 4: In this table we compare evasion mitigation approaches. We note a general evolution away from mitigations which counter specific evasions towards more general approaches based on inherently more transparent technologies.

Tool	Evasion Technique Mitigated	Method	Year
Cobra [167]	Known anti-analysis techniques	Dynamic code rewriting	2006
HoneyMonkey [174]	General (web)	Bare metal analysis	2006
CWSandbox [177]	Environmental sensitivity	Rootkit-like code hooking	2007
Moser et al. [118]	Conditional execution	Forced path exploration	2007
MineSweeper [29]	Trigger-based behavior	Forced path exploration	2008
Ether [46]	Emulator & hosted VM detection	Hypervisor (Xen) with system call monitoring	2008
Kang et al. [86]	General	Dynamic state modification	2009
MAVMM [124]	General	Hypervisor (custom)	2009
jsand [40]	General (JavaScript)	Forced path exploration	2010
BareBox [94]	General	Bare metal analysis (kernel driver)	2011
V2E [183]	Anti-emulation	Multi-system record and replay	2012
Rozzle [102]	Environment sensitivity (JavaScript)	Forced path exploration	2012
NVMTrace [147]	General	Bare metal analysis	2012
Spectre [191]	General	Bare metal analysis (x86 SMM observation)	2013
DRAKVUF [107]	General	Hypervisor (Xen) with injected breakpoints	2014
Alwabel et al. [8]	General & CPU red pills	Dynamic state modification	2014
BareCloud [96]	General	Bare metal analysis (external disk and network monitoring)	2014
A5 [170]	General (Android)	Bare metal analysis	2014
BareDroid [121]	General (Android)	Bare metal analysis	2015
MalT [190]	General	Bare metal analysis (x86 SMM debugging)	2015
Hops [106]	General	Bare metal analysis (x86 SMM and PCI)	2016
LO-PHI [156]	General	Bare metal analysis (RAM and disk capture)	2016
Gordon [180]	General (Flash)	Forced path exploration	2016
J-Force [92]	General (JavaScript)	Forced path exploration	2017

including malware. Ugarte-Pedrero et al.'s [162] work on RAMBO proposes optimizations to increase the efficiency of path exploration-based binary unpacking.

Flash: Wressnegger et al.'s [180] work on Gordon proposes a path exploration algorithm for use against Flash malware. While the authors do not cite any evasive Flash malware observed in the wild, they note the possibility and discuss potential methods of detection and mitigations.

4.4 State Modification

As discussed in §3.2.1, Kang et al. [86] propose mitigating evasion with "dynamic state modification" (DSM), referring to modification of program execution state (i.e., registers or memory) at given points during execution. After identifying a root point of evasion, in order to make execution proceed further DSM can be used to force code to take alternate branches. Kang et al. evaluated their work by testing six evasive malware samples; the system was able to create DSMs to mitigate evasion in all six.

Alwabel et al. [8] propose hiding CPU emulator red pills by freezing emulator state after each instruction and correcting system state where it diverges from expected state on a real system.

4.5 Multi-Platform Record & Replay

Yan et al.'s [183] V2E uses "precise heterogeneous replay" on virtualized and emulated systems to enable transparent malware analysis. While Kang et al. [86] use instruction traces from multiple platforms to detect anti-analysis and produce DSMs, Yan et al. point out that this approach is not scalable to malware with multiple anti-analysis checks in different execution paths. Rather than aiming to identify and subsequently mitigate evasion, V2E seeks to simply make it irrelevant through transparent analysis.

V2E collects information about malware execution within KVM[26] and then replays executed code under analysis in Temu [185]. By decoupling recording and analysis, the system takes advantage of KVM's relatively greater transparency, while also using Temu's rich binary instrumentation.

The researchers used V2E to analyze fourteen emulation-evasive malware samples and succeeded in analyzing all of them, extracting code and data only accessible after unpacking.

4.6 Hypervisor-based Analysis

Researchers have proposed using hypervisors for analysis due to their relatively greater transparency, though these systems have since been shown to be detectable [26, 130, 131].

Dinaburg et al.'s [46] Ether monitors malware by catching system calls and context switches using Xen. In evaluation, the authors analyzed 25,118 malware samples, some of which were evasive. Testing an Ether-based unpacker against fifteen packers which evaded Renovo [85][27] and PolyUnpack [51],[28] the system succeeded in unpacking malware packed with all of them. An Ether-based program tracer was successful in tracing malware packed with eighteen different packers, a few of which Anubis [19] and Norman [4] were unable to successfully trace. Despite the lengths Dinaburg

[26] A hardware-supported virtual machine [98]
[27] A QEMU-based malware unpacker built on top of BitBlaze [155]
[28] A VMware-based malware unpacker

et al. went to in order to make Ether transparent, Pék et al. [130] were able to demonstrate timing and CPU-based detection attacks against Ether with nEther.

Lengyel et al.'s [107] DRAKVUF uses Xen, but instruments code with injected breakpoints rather than only logging system calls like Ether. In evaluation, the authors analyzed 1,000 malware samples suspected to exhibit evasive behavior.

Nguyen et al.'s [124] MAVMM uses AMD SVM to support a *custom* hypervisor purpose-built for malware analysis. In testing, the authors used the hypervisor to log system calls made by a Linux malware sample, and demonstrated its resilience against various VMM-detection techniques. Thompson et al.'s [160] technical report demonstrates timing attacks that can be used to detect MAVMM and other virtualized systems.

4.7 Bare Metal Analysis

In response to the challenges of transparent virtualization, some researchers have turned to *bare metal* malware analysis, where malware is run without virtualization on hardware with internal or external system instrumentation.

4.7.1 PC. While they did not propose a full bare metal analysis system, we note Chen et al.'s [36] early work on evasive malware, which ran samples in VMWare, and on bare metal systems and bare metal systems under debuggers. Testing 6,222 malware samples, the authors found that 2.7% exhibited less malicious behavior in VMware than on bare metal.

Kirat et al. [94, 96] proposed two bare metal analysis systems, BareBox and BareCloud. With their work on BareBox, the authors used an in-system kernel module to monitor malware execution, though this could obviously be detected as an environmental artifact by malware. In evaluation of BareBox, the authors collected 200 malware samples across seven malware families known to detect virtualized analysis systems and executed them within VMware, QEMU, and BareBox. Every malware family exhibited more network activity and process creation within BareBox than the virtualized systems.

As discussed in §3.2.4, BareCloud's bare metal-analysis system analyzes disk and network activity at the hardware level without the use of a kernel module. The system's increased transparency (relative to BareBox's in-system analysis) comes as the cost of introspection ability, as stalling code cannot be detected, nor can file modifications which do not change size and time metadata (a limitation of the SleuthKit framework used to analyze disk activity).

Royal's [147] presentation and whitepaper proposes NVMTrace, a bare metal analysis system which provides analysts with access to network traffic and disk contents to be analyzed with forensic tools.

Willems et al. [178] propose using hardware branch tracing to transparently track code execution. In experimentation, the authors analyzed an evasive PDF and 4,869 malicious (but not necessarily evasive) PDFs.

Spensky et al.'s [156] LO-PHI instruments physical hardware in order to enable bare metal malware analysis using open source analysis tools such as Volatility and Sleuthkit. LO-PHI captures RAM and disk activity during malware execution, and enables scriptable simulated user interaction through a USB-connected Arduino that emulates mouse and keyboard input. In experimentation with evasive malware, the authors were able to analyze Paranoid Fish [6][29] without being detected. Additionally, analyzing a set of over 250 evasive malware samples collected by Kirat et al. [96], LO-PHI successfully captured information about malware behavior. The authors noted that they were unable to definitively assess the system's efficacy due to lack of ground truth about behavior, a network setup blocking external communication, and binaries targeting different operating systems.

4.7.2 PC - SMM-Based Analysis. Recent work has turned to low-level x86 system management mode (SMM) to enable bare metal analysis. Zhang et al. [191] point out several traits of SMM that make it useful in bare metal analysis including full access to system memory, inability to be modified once started, high speed, and protection from modification.

Zhang et al.'s [191] Spectre uses SMM introspection to examine (but not modify) running code 100 times faster than similar VMM-based systems. Zhang et al.'s [190] further work on MalT uses SMM-based *debugging* for malware analysis, providing analysts with rich insight into system activity.

Leach et al.'s [106] Hops uses SMM to capture live memory snapshots and also supports PCI-based instrumentation. The authors evaluated the tool by capturing stack traces from Paranoid Fish [6], which were then analyzed to detect of 16 of the tool's 22 evasion techniques.

4.7.3 Android Analysis. Mutti et al.'s [121] BareDroid brings bare metal analysis to Android malware. The authors' approach to bare metal Android analysis relies on using fast system restoration between malware executions to revert modifications made to system drive partitions. The system monitors malware execution by logging information about file operations through SELinux. In evaluation the authors tested nine evasive Android apps and found that all nine exhibited additional behavior not observed while running in emulators.

Vidas et al.'s [170] A5 uses virtualized and bare metal analysis systems in order to obtain network-based malware indicators. The authors' focus was a novel static and dynamic analysis-based approach to Android malware analysis rather than innovations in bare metal analysis, so they did not evaluate the efficacy of the approach versus virtualized analysis.

4.7.4 Browser Analysis. Wang et al.'s [174] early report on their HoneyMonkey web malware analysis systems notes the possibility of of VMM-evasive browser-based malware. In response to this challenge, the researchers developed techniques for running their analysis on bare metal Windows systems using system checkpoints and copy-on-write disks. Unfortunately, the authors did not evaluate the efficacy of bare metal versus VMM-based HoneyMonkey systems.

4.8 Discussion

In this section, we have surveyed over a decade of work on malware evasion mitigation. The problem of evasive malware is likely to continue to challenge defensive researchers in the future. In table 4

[29] An open source tool for testing analysis systems by looking for common fingerprints

we compare approaches to mitigation, showing trends in research turning towards approaches based on inherent transparency rather than detection-enabled mitigation.

Evasion mitigation approaches can be broadly divided into two categories: active approaches involving detection and subsequent mitigation, and passive approaches based on increasing analysis system transparency. We find that passive efforts based on transparent analysis (e.g., hypervisors and bare metal analysis) have been more prevalent than active approaches. We believe that the attention these techniques have received is due to their better scalability compared to detection and subsequent mitigation, or path exploration-based tactics, as both are vulnerable to denial of service attacks, e.g., through the use of multiple evasion techniques or path-exploding obfuscations.

Several recent works have turned to bare metal analysis as a mitigation for virtualization-related fingerprints, but bare metal is not necessarily a panacea against fingerprinting, as these systems remain vulnerable to an array of attacks (all except for those related to CPU emulation).

5 CASE STUDIES

In this section, we survey works on attacks against deployed analysis systems, and research on evasion in wild malware.

Note that as with rest of this survey, we only cover academic offensive and defensive works, so discussed statistics may not be representative of the threat landscape observed in industry. Offensive actors creating malware are likely to be more aggressive in conducting fingerprint discovery efforts than academic researchers. Defensive assessments of malware evasion are likely better understood by industry antivirus and endpoint protection companies which face evasive malware daily.

5.1 Offense - Attacking Deployed Systems

While attackers can experiment with fingerprinting underlying virtualization platforms in offline lab environments, it can be difficult to discover fingerprints introduced as a result of system configuration or analysis instrumentation. In this section we analyze "active reconnaissance attacks"[30] in which researchers have exploited network connectivity or analysis reports to exfiltrate fingerprints from analysis systems for use in evasive malware. These attacks can give attackers valuable information about the defensive threat landscape and help them discover system-specific fingerprints, but can also increase risk of future compromise.[31]

5.1.1 PC. Yoshioka et al.'s [187] work has demonstrated that the threat of enabling network connectivity in analysis systems extends beyond simply allowing malware authors to find out that their malware is under analysis. The authors propose uploading malware to publicly accessible analysis systems where it can collect environmental fingerprints such as product IDs, files, registry keys, or running processes, and exfiltrate them to an attacker controlled server. In evaluation, the authors were able to fingerprint and evade fifteen targeted systems.

[30] We take this term from Kirat et al.'s [96] description of Yoshioka et al.'s [187] work.
[31] e.g., compromising their IP addresses, tactics and techniques, or malware signatures. Defenders running these systems may be able to create signatures for early builds of their malware that can be used to detect subsequent builds [90]

Yokoyama et al.'s [186] SandPrint builds upon Yoshioka et al.'s exfiltration technique, but takes a more nuanced approach to fingerprinting, looking for traits likely to be associated with automated analysis systems, such as small screen resolutions, single core CPUs, small amounts of RAM, and long durations since last login or boot. The authors submitted SandPrint to twenty malware analysis services and collected a total of 2,666 analysis reports which they used to cluster 76 distinct automated analysis systems.

Brengel et al. [26] evaluated timing-based Intel VT-x detection attacks against seventeen online automated analysis systems and exfiltrated their findings to a server under their control. Of the 76 reports the authors received, 74 were from systems supporting VT-x, and their detection attacks succeeded against all 74.

Rolles' [143] blog post describes an attack on Renovo [85], where he used JITted code detection to extract fingerprints from the system.

5.1.2 Android. Oberheide and Miller's research [126] used reconnaissance attacks to fingerprint Google's "Bouncer" system for analyzing Android apps submitted to the Google Play Store. The researchers proposed a number of potential fingerprints and fingerprinted Bouncer's environment, IP address, and user interactions. Numerous subsequent researchers have similarly exploited network exfiltration to fingerprint Android analysis systems [24, 45, 64, 81, 112, 113, 133, 134, 169].

5.1.3 Antivirus. Blackthorne et al.'s [21] AVLeak enables fingerprint discovery against consumer antivirus emulators. Exploiting malware detections as a side channel to leak information from within emulators, the authors discovered hundreds of fingerprints in four antivirus products. The researchers also discovered antivirus emulator-related fingerprints in many wild malware samples, including one from an APT.

5.2 Defense - Quantifying Evasion

Several papers have taken on the challenge of analyzing evasive behavior in large-scale studies of malware found in the wild. While useful in characterizing the real-world threats posed by evasive malware, these studies may have their statistics distorted by factors such as advanced malware which evades deployed anti-analysis mitigations, the difficulty of catching malware in the wild, and non-evasion related environmental factors in malware behavior (e.g., unavailability of command and control servers at the time of analysis, operating system configuration, discrepancies in malware invocation, etc.).

5.2.1 PC. Chen et al. [36] proposed a taxonomy of anti-analysis techniques used by malware in the wild and found that 2.7% of 6,222 samples from 2006-2007 exhibited less behavior when run in a VM versus a bare metal system.

Bayer et al.'s [103] 2009 work evaluates malware behaviors in almost one million samples analyzed in Anubis. The authors found that 0.03% of the samples exhibited a specific anti-Anubis check (i.e., for specific Anubis environmental artifacts), but they were unable to assess the prevalence of more subtle checks, such as those as the CPU level.

Chen et al. [34] surveyed anti-VMM and anti-debugging techniques in 17,283 "generic" and targeted (APT) malware samples collected between 2009-2014. The authors found that anti-analysis techniques became more prevalent over the surveyed timeframe, and that their presence was negatively correlated with antivirus detection. The authors also observed that APTs generally do not use as many anti-analysis techniques as generic malware, and that they have decreased their use of these techniques over time.

5.2.2 Web. Rajab et al.'s [138] 2011 technical report evaluates four years of data on evasion in web-based malware. We refer interested readers to their paper, as the results are too numerous to reproduce here.

Kapravelos et al.'s [88] 2011 work discusses attacks on browser analysis systems seen in the wild, and briefly evaluates the prevalence of evasive web malware, finding a "significant chance" of anti-analysis in 8,835 of 33,557 URLs examined.

5.3 Discussion

In this section we have analyzed attacks on malware analysis systems in the wild, as well as efforts to assess the prevalence and nature of evasion in wild malware.

While surveyed academic researchers have mounted active reconnaissance attacks against analysis systems in the wild, we note that it is challenging to assess how malware authors discover the fingerprints that they use for evasion. It is likely that malware authors use these style of attacks [21, 103], but we are not aware of any works analyzing how these actors discover their fingerprints; future researchers may look to answer this question. Future research on wild malware behavior should note if any samples use fingerprints against specific systems only discoverable through active reconnaissance or other system penetration.

We have mainly focused on surveying academic research, but we note that in the wild, malware authors face a wide range of threats including antivirus and high-end enterprise protection systems. Malware authors likely prioritize evading these systems and the immediate threat of industry signaturing and collection over evading academic research systems.

6 DISCUSSION

In this section we discuss challenges to research evaluation and propose future topics in offensive and defensive research.

For all of the research we have covered from the past decade, we believe that the field has ultimately not moved forward in any *fundamental* ways. Malware still detects and evades analysis, and defenders still constantly propose novel mitigations against evasion. As in many defensive computer security fields, a constant battle between attacker and defenders is to be expected. However, unlike other fields which have succeeded in nigh completely eradicating classes of attacks, or making attackers significantly change their tactics,[32] defensive researchers have not stopped malware from using conditional evasion, or eliminated the use of any particular class of fingerprints in wild malware.

Research has offered piecemeal solutions for individual evasion techniques, but these technologies may come at significant costs to scalability, or may mitigate a few classes of fingerprints while leaving others unaddressed. Despite our pessimism about research efforts thus far, we note that researchers in this field face *incredible* challenges from overwhelming amounts of malware, difficulty in experimentation, and fundamentally difficult, if not undecidable problems.

6.1 Research Evaluation

In surveying works on evasive malware, we noted extreme variation in rigor of evaluation. Surveyed works range from analyzing single malware samples to analyzing millions.[33] Works also vary significantly in their evaluation methodologies, from simply analyzing samples without comparison to other approaches; to comparing data to results from ostensibly less transparent systems; to testing against known corpora of samples with ground truth produced by human analysts. This broad spread of evaluation rigor makes it extremely difficult to empirically compare the efficacy of various approaches to detecting and mitigating evasion. Throughout this paper we have reproduced statistics from surveyed works, though due to the challenges of comparing results, we do not empirically rank them.

We are not alone in noting weaknesses and challenges in evaluation of malware research. Rossow et al.'s [145] paper evaluates the "methodological rigor and prudence" of 36 publications involving malware experimentation from 2006-2011, finding problematic assumptions about datasets in 25%, and lack of security precautions in 71%. Kantchelian et al.'s [87] paper notes challenges in establishing ground truth in malware labeling using antivirus software, while Hurier et al.'s [76] work takes on the same issue for Android malware.

6.1.1 Suggestions For Improvement. Rossow et al. [145] offer a number of suggestions for improving the quality of malware experimentation; we refer readers to their paper for more details. In addition to reemphasizing all of Rossow et al.'s suggestions, we offer several specific to experimentation with *evasive* malware.

Establish Ground Truth: As we discuss in §6.3.2, it can be extremely difficult to establish ground truth about evasive malware behavior. However, in experimentation, researchers should attempt to have ground truth (e.g., from a human malware analyst) for at least *some* of the malware under test.

Make Multi-Execution Systems Similar: When conducting experiments involving running malware multiple times in different environments (e.g., virtualized versus bare metal), differences unrelated to the evasion techniques being tested should be minimized (e.g., file system contents, time, settings, hardware resources, etc., should all be the same) so that spurious differences in execution are not conflated with evasion. Any unavoidable differences in execution environments should be noted and discussed with regard to their potential effect.

Be Explicit About Malware Origins: Researchers should note when and where they obtained malware for experimentation. By its very nature, evasive malware seeks to avoid detection, so collecting test corpora is a challenging task. Corpora assembled from different sources may reveal different biases, e.g., malware collected from upload-based sites such as VirusTotal may contain samples found in the wild by researchers as well as samples uploaded by malware

[32] e.g., the eradication of format string vulnerabilities, or the introduction of DEP and ASLR in anti-exploitation mitigations

[33] Some have even used synthetic researcher-created samples, though most surveyed works use "wild" malware.

authors seeking to test their malware against AVs; web malware sets gathered by web crawling may not contain targeted attacks; etc. As Bayer et al.'s [103] work has shown, malware authors may evolve their malware to evade analysis systems over time if they are left accessible and open to reconnaissance attacks.

6.2 Offensive Research

In this section we discuss directions in offensive research that may be interesting to academic researchers and malware authors.

6.2.1 Evolving Fingerprinting. We propose several topic for continued research on analysis fingerprinting.

Reverse Turing Tests: Defensive research on building transparent analysis systems and analyzing evasive malware has generally looked to addressing the underlying technological issues of virtualization and analysis instrumentation detection. Comparatively little research has gone into mitigating anti-analysis in user-sensitive malware (likely due to its relative rarity in the wild given the ease of implementing virtualization detection attacks). As analysis systems become increasingly transparent, we believe that advanced malware will look to user activity-related fingerprints to evade detection. We highlight several works that we believe point in promising new directions for reverse Turing Testing.

As discussed in §5.1.1, Yokoyama et al.'s [186] SandPrint highlights the potential for fingerprinting based on subtle traits indicative of analysis systems' lack of a human user. Miramirkhani et al.'s [116] work has shown how subtle "wear-and-tear" artifacts can lead to passive identification if a real human has used the system.

Diao et al.'s [45] PIDetector aims to hinder automated analysis by detecting simulated input events through unique metadata associated with inputs, as well as the unchanging regularity of these events (e.g., touch screen movements in perfect straight lines carried out at constant speed). PIDetector also enhances applications with hidden UI elements and application activities unlikely to be noticed or interacted with by a human, but easily discovered through programmatic UI parsing APIs (e.g., invisible, off screen, or tiny controls). When these hidden UI elements are interacted with, the application can detect that it is being analyzed.

As Dolan-Gavitt and Nadji [49] point out, offensive researchers and malware authors may look to work on detecting automated cheating "bots" in online gaming (e.g., through varying speed, accuracy, and duration for mouse and keyboard inputs) for inspiration in human user detection [66, 84]. Researchers may also draw inspiration from work on detecting online bots by their interaction patterns [38, 80].

Fingerprinting Browsers: Researchers seeking to evade web browser analysis systems may draw upon work on browser fingerprinting for user tracking online. A number of researchers have looked at the possibility of fingerprinting specific web browsers, and while this research has mainly considered the privacy concerns for users, we note the obvious implications for web-based malware evasion. We refer readers to Eckersley's [50] seminal paper on the topic, and the numerous subsequent papers which have built upon it.

Detecting Bare Metal Analysis: As the state of the art in automated dynamic malware analysis moves to bare metal analysis, it will be increasingly important for attackers to develop anti-analysis techniques that do not rely on detecting virtualization. Malware authors may look to advanced stalling techniques (due to the difficulty of detecting them without close introspection capabilities [96]), active reconnaissance attacks against bare metal systems, and user activity or "wear-and-tear"-based artifacts [91, 186].

Offensive researchers may also look to attacks against the unique characteristics of bare metal analysis systems. Spensky et al. [156] note the possibility of detecting LO-PHI through timing discrepancies for RAM and disk accesses.

Researchers may also look to ways of detecting instrumented or scripted peripherals used to give input to bare metal systems (e.g., Spensky et al.'s [156] use of an Arduino to emulate a keyboard and mouse).

6.2.2 Fingerprinting Alternative Platforms. Future research may look to fingerprinting and evading alternative analysis platforms, including Flash [180], antivirus [21], IoT [151], BIOS [69], and PDF [178]. Researchers may also develop methods of detecting SMM-based analysis systems, which have been claimed to be highly transparent [191].

6.2.3 Deception. Surveyed works have generally addressed malware which seeks to evade detection by aborting execution after detecting analysis. As the very act of terminating execution prematurely may be a heuristic indication of malicious intent, future malware authors and offensive researchers may look to deception attacks - misleading analysis systems instead of simply denying them insight.

Rather than exiting, future malware authors may explore the possibility of creating false positives by pretending to be commonly known or old malware. This evasion technique was used by `Tilon`, a financial trojan, which drops a fake "scamware" tool if it detects virtualization [99]. Similarly, Bulazel [21, 30] has demonstrated the possibility of deceiving antivirus emulators by dropping uninteresting MS-DOS viruses in order to prevent additional scrutiny associated with heuristic malware detection.[34] Vigna [171] has discussed the possibility of malware which mimics benign software when it detects that it is being analyzed.

6.3 Defensive Research

In this section, we discuss defensive challenges for future research in analyzing and experimenting with evasive malware.

Defensive researchers are faced with immense challenges, most notably the ultimately undecidable problem of the deciding if a given program may ever reach a given state. Defenders are also constantly challenged by novel attacks and the overwhelming constant deluge of new malware. Unlike many other cybersecurity fields, defensive malware researchers don't face abstract challenges in systems security or cryptography, but literally fight on the front lines to detect and disrupt attacks by criminals and state-sponsored intelligence and military services [70, 141]. These attackers are not only competent professionals, they are also likely better resourced than defenders.

6.3.1 Improving Bare Metal Analysis. While a number of bare metal malware analysis systems have been recently proposed, these

[34] i.e., detection based on an unknown binary's suspicious traits or actions that could lead antivirus vendors to investigate it more deeply

systems suffer from obvious issues of scalability compared to VMM-based alternatives. Bare metal analysis is challenged by limitations in power consumption [121], physical hardware life [147], and system restoration between runs [94, 96]. Future researchers may look to making bare metal analysis scalable, including using both novel hardware designs and improvements in analysis efficiency [18, 79].

Efficient Bare Metal Analysis: Vadrevu and Perdisci [163] propose MAXS, a network traffic event analysis-based system that can determine if a malware sample under analysis has been seen before (i.e., the same underlying binary with a novel packing or obfuscation scheme), so that bare metal systems can abort analysis early, conserving precious resources. In evaluation the authors were able to reduce execution time by up to 50% on average, with less than 0.3% information loss. Future researchers may continue this line of inquiry in order to increase the efficacy of analysis systems, bare metal or otherwise.

Improving Introspection: Bare metal analysis' transparency often comes at the cost of insight into malware behavior [96]. Zhang et al. [190, 191] and Spensky et al.'s [156] recent works have proposed new approaches to conducting bare metal analysis without sacrificing introspection ability. Additional research into this area is important, as bare metal systems show great promise for the future of automated dynamic malware analysis.

Mitigating Stalling: Bare metal analysis systems are particularly vulnerable to stalling attacks [96]. Without visibility into system calls or execution progress, it may be particularly difficult to assess if malware is stalling. We propose that performance tracing processor features (e.g., Intel PMU and PT) may be one way of detecting stalling behavior while keeping analysis on bare metal, though with in-system instrumentation (a kernel driver managing the performance tracing) [7, 178].

6.3.2 *Establishing Ground Truth.*

Efforts to establish ground truth for experimentation face numerous challenges. The entire endeavour of detecting and mitigating evasive behavior is fraught with *unknown-unknowns* - researchers don't know what they don't know about malware evasion.

Manual human malware analysis is not scalable for establishing ground truth for large-scale malware corpora, though it may be used on subsets of these corpora [96]. "Bootstrapping" malware corpora by building ground truth from automated analysis reports is problematic due to issues of command and control, randomization, and targeting; as well as potential spurious execution differences due to the specifics of system setup. Collecting malware in the wild poses its own challenges, particularly when looking for *evasive malware*, which by nature attempts to avoid detection. Collection systems such as honeypots or upload-based sites are unlikely to catch APT attackers, who may go undetected in the wild for years, and have been known to occasionally employ anti-analysis [16, 21, 34].

Dolan-Gavitt et al.'s [48] recent work on LAVA has sought to create *synthetic* ground truth corpora on binary program vulnerabilities and exploits, another field in which establishing ground truth is extremely challenging [5] (perhaps even harder than evasive malware). While synthetic corpora or test cases such as Paranoid Fish [6] are not necessarily representative of evasive behaviors seen in the wild, they may be a step forward in enhancing experimental rigor. Researchers looking at corpora synthesis could draw upon work in automated fingerprint discovery, such as Jing et al.'s [81] Morpheus.

6.3.3 *Heuristic Evasion Detection.*

Academic research on automated malware analysis has generally looked to enhancing the clarity and depth of analysis in order to better understand malware behavior. In contrast, industry and consumer anti-malware products prioritize preventing damage or infection, and may outsource difficult analysis problems to expert human analysts or advanced tools after stopping malware execution. Future research may look to *heuristic* methods for detecting evasion in order to preemptively detect malware execution. Researchers may draw on ideas such as such as Kolbitsch et al.'s [102] insight that code *fragility* is often a sign of maliciousness, heuristic malware detection in antivirus emulators based on process introspection behaviors [21, 30], Hasten's [101] stalling detection based on system call patterns, MAXS' [163] techniques for detecting malware from event traces, and Polino et al.'s [135] discussion of "easily" detecting "weird" and "inefficient" memory allocation-based anti-DBI techniques.

6.3.4 *Passing Reverse Turing Tests.*

In order to foil advanced malware using reverse Turing Tests to evade detection (as discussed in §6.2.1), defensive researchers should look to ways of believably simulating human presence on automated analysis systems.

We note Kharraz et al.'s [91] work on creating realistic execution environments for experimentation with ransomware for UNVEIL. In order to prevent ransomware from detecting that it is running in an automated analysis environment, UNVEIL goes to great lengths to create fake file systems, generating files with valid headers, using a database of sentences to generate plausible documents, synthesizing file names from a word list, and creating believable directory structures and file timestamps. Future researchers may look to applying techniques like these to generating other "wear-and-tear" artifacts beyond just file system contents.

Rossey et al.'s [144] work on the LARIAT information assurance testbed may provide inspiration for automating common user behaviors such as web browsing and authoring documents which malware may look for as indicators of active human use of a system.

Future research must also address the challenges of malware that looks for active human interaction such as mouse or keyboard input. As Diao et al.'s [45] work on PIDetector for foiling *Android* automated analysis systems has shown, systems faking user input can be detected if the input is too machine-like. Research looking in this direction may draw inspiration from the field of biometric spoofing [9].

6.3.5 *Game Theory Formalizations.*

We propose that formalized rigorous approaches to modeling malware and analysis system interactions may be helpful in advancing the field.

Analysis evasion and evasion detection can ultimately be formulated as a "cat-and-mouse" game where malware chooses an evasion strategy, while an analysis system responds with an evasion detection strategy. In one round constructions, the game is over when the analysis system wins by identifying the malware's evasion, or loses by failing to detect it.

It is possible that an analysis system or malware sample may adapt a "no-operation" strategy of not mitigating evasion or attemping to evade analysis unless some predetermined condition is satisfied. For example, use of a particularly sophisticated or unknown ("zero-day") evasion technique or evasion mitigation may be conditional upon analysis of, or execution within, a specific adversary or verification that adversary is "worthy." Testing such a condition may also be thought of a fingerprint in its own right.

Choosing a no-operation tactic can be thought of a suboptimal strategy (since it intentionally precludes successful exploitation); however, it opens up an interesting direction for adversarial learning research, in which a training set may be constructed in such a way that a "learner" learns an incorrect function.

Another interesting formulation would involve casting the problem as a Stackelberg game [28, 173] where an analysis system (defender) allocates some resources (e.g., for emulation) as a leader, and malware (attacker) plays as a follower. The analysis system can randomize its strategy (e.g., in response to malware action) while the analyzed malware deploys a purely deterministic evasion strategy.

7 RELATED TOPICS

In this section, we briefly survey a variety of related topics in malware evasion and automated analysis.

7.1 Static Anti-Analysis

Static anti-analysis is an incredibly broad field and includes techniques such as anti-disassembly [14, 78], general code obfuscations [37, 119, 152, 188], virtualized obfuscators [142], and packers [27, 146, 161, 184].

7.2 Anti-Debugging

Malware can use anti-debugging to hinder human debugger-based analysis. We note Chen et al.'s [36] analysis of anti-debugging in the wild, Gagnon et al.'s discussion of anti-debugging for software protection, [63] Branco et al.'s [14] survey of anti-analysis in four million malware samples, and Shields' [153] anti-debugging implementation guide. Defensive researchers have developed specialized debuggers to mitigate anti-debugging [43, 190].

7.3 Software Protection

While we have surveyed uses of anti-analysis techniques in *malicious* software, we note that these techniques may also be used by legitimate non-malicious software vendors in order to prevent analysis or piracy of their software. Software protection schemes are more likely to include anti-debugging, obfuscation, and anti-static analysis protections than the conditional evasion techniques surveyed here [21, 39, 63]. Analysis or piracy detection conditional behaviors have been implemented in some video games, blocking player progress if detected [58].

7.4 Non-Conditional Evasion

Our survey has looked at malware which evades dynamic detection through conditional code execution. Attackers may also create malware which evades detection *without* acting conditionally. While it is difficult to empirically measure the prevalence of these techniques in wild malware, our intuition from experience and literature survey leads us to assess that these techniques (other than stalling) are not widely used outside of research, as they are more difficult to implement than conditional evasion and less reliable.

7.4.1 Stalling. Stalling allows malware to evade automated analysis without fingerprinting or conditional execution, circumventing advances in transparency, and foiling efforts to detect conditional evasion (as stalling code is executed unconditionally). Malware may stall execution by using long-running loops, instructions which slow down virtualization, operating system "sleep" calls, or calls to other time consuming operating system functions, among other techniques [88, 101, 113, 174]. Advanced malware may use the results of stalling computation in later parts of execution (e.g., deriving a cryptographic key as the result of many time-consuming operations) in order to force analysis systems to actually execute stalling code [101].[35]

Kolbitsch et al.'s [101] work on HASTEN proposes a method of detecting and mitigating stalling. HASTEN monitors system calls made by a piece of malware as a means of measuring execution progress, and uses heuristics to detect stalling. The system is able to mitigate the effects of stalling passively through reduction of analysis instrumentation overhead and actively through modification of program control flow.

7.4.2 Environmental Keying. Environmentally keyed malware uses cryptography to encrypt itself with keys derived from environmental traits of its target system (e.g., hardware serial numbers, file names, environment settings, etc.), or relies on an external server to provide keys once the malware makes contact from a target system [22, 189]. As Song and Royal [154] point out, "if widely adopted by the criminal underground, [environmentally keyed malware] would permanently disadvantage automated malware analysis by making it ineffective and unscalable." We refer readers interested in learning more about environmental keying to Morrow and Pitts' [117] work, which presents an annotated bibliography of prior research and malware in the wild.

7.4.3 Analysis Subversion. Malware may actively subvert analysis by misleading or attacking instrumentation.

Attacking The Host: In virtualized systems, malware may attack the analysis system it is running within, using vulnerabilities in VMMs to crash them or break out to infect host systems [21, 127, 131].

Anti-VMI: Attackers may mislead systems that use virtual machine introspection (VMI) by modifying structures used to track machine state. Bahram et al.'s [11] direct kernel structure modification modifies kernel data to subvert VMI.

Anti-Taint: Malware may use techniques to subvert taint tracking in taint-analysis-based systems [33]. Taint analysis is particularly challenging for Android [109], and attacks against these systems have been demonstrated [10].

[35] If stalling code does not actually affect subsequent control flow (e.g., a long sleep call, or a loop that increments a value from zero to a large number without using the value after executing), it can be skipped over without impacting the analysis system's ability to observe the malware's behavior.

System Specific Attacks: In addition to these more general techniques, malware may attack specific analysis systems on a case-by-case by removing, bypassing, or exploiting their instrumentation [55, 125].

7.5 Trigger Detection & Mitigation

Some malware exhibits *trigger-based behavior*, where it acts in response to input-based triggers, such as time, location, or network activity. Trigger-based behavior may be used for analysis evasion, but may also simply be part of malware design (e.g., malware that waits for a user to type a credit card number into a form, or "logic bombs" that wait to act until a given time or date) The study of trigger-based malware is broad, and we cannot comprehensively capture it here, though future researchers may find work in more systematically surveying the topic.

7.5.1 PC. As discussed in §4.3, Moser et al. [118] and Brumley et al.'s [29] works use forced path exploration to analyze malware, including trigger-based malware. Crandall et al.'s [41] research looks specifically at time-triggered malware.

7.5.2 Android Events. A number of papers have looked to analyzing Android malware with event-based triggers, including user interface triggers not easily triggered with random input [192], event injection based on static analysis [159]; injection based on detecting use of APIs of interest [179], human-like user interface interaction [32], and automated trigger discovery for "logic bomb" apps [61].

7.5.3 User Input Triggers. In this section we briefly survey works on analyzing user input triggered malware, which malware acts in response to interaction or input from a user.

PC: Holz et al. [72] propose SimUser, a user interaction simulator for use in analyzing keyloggers through AutoIt-based stimulation. Fleck et al.'s [59] PyTrigger records user inputs and system context while performing potentially triggering actions, and replays these inputs during analysis.

Android: Suarez-Tangil et al.'s [157] work seeks to detect Android malware that targets specific users by using stochastic models of usage from traces of real users. Elish et al. [53] propose a method for detecting how user input may result in sensitive API function invocations in Android apps.

Web: Kirda et al.'s [97] work generates web browser events simulating user interaction in order to analyze Internet Explorer Browser Helper Objects. Testing their analysis system, Rajab et al. [138] noted a 40% increase in downloaded binaries when adding mouse clicking on pages.

7.5.4 Network Simulation. While working with network-triggered malware, researchers have proposed simulating network connectivity to isolate the malware while also allowing insight into its behavior. Systems simulating network connectivity may return generic protocol success messages for communications, conditionally proxy network connectivity, or use program analysis to solve for network inputs that trigger malware behavior [8, 21, 67, 136].

7.6 Other Topics

In this section we collect other related topics in anti-analysis.

7.6.1 Discovering Targeted Systems. Xu et al.'s [182] GoldenEye seeks to identify target environments for *environmentally-targeted* malware, which chooses to run based on its execution environment.[36] Johnson et al. [82] use differential slicing to capture input differences that trigger execution path differences.

7.6.2 Remote Detection. Beyond in-system attacks, remote detection of virtualization is also possible. Kohno et al. [100] and Franklin et al. [60] propose remote attacks based on in-VM timing overhead. Chen et al. [36] demonstrate remote fingerprinting against VMware and Xen using TCP timestamps.

7.6.3 Harvesting Runtime Values. Rasthofer et al.'s [139] Harvester aids human analysts by collecting runtime-calculated values from Android apps. Using program slicing to extract code relevant to value derivation, Harvester can remove conditional evasive code and extract calculated values.

7.6.4 Vaccination. Chen et al. [36] propose deterring evasive malware from running on regular systems by making them appear to be analysis systems with fake environmental artifacts and function hooks. Researchers have also looked at "vaccinating" systems by faking malware infections to deter malware which does not run on already infected hosts [176, 181].

7.6.5 Formalizations. Fu [62] proposes a formal theoretical model of environmental sensitivity for programs, and presents a slicing model to characterize environmentally sensitive program gadgets. Blackthorne et al. [23] propose a formalized theoretical model for the "observer effect" caused by analysis.

7.6.6 Anti-DBI. Researchers have looked to detecting and evading dynamic binary instrumentation tools which are used in debugging and automated analysis [74, 110, 133, 135, 158]. Polino et al.'s [135] recent work on anti-DBI found that 15.6% of 7,006 malware samples collected from VirusTotal between October 2016 and February 2017 exhibited anti-instrumentation behavior (not necessarily solely anti-DBI).

7.6.7 Virtual Machine-Based Rootkits. Offensive researchers have proposed virtual machine-based rootkits [149]. These rootkits generally use hardware supported virtualization to hijack and virtualize a guest operating system [42, 44, 93]. Researchers have also exploited SMM and other chip features to create even lower level rootkits [54, 150]

7.6.8 Fileless Malware. Researchers and malware authors in the wild have created fileless malware not based on actual executable binaries stored on disk.

WMI: Windows Management Interface (WMI)-based malware essentially exists as a set of small scripts (e.g., PowerShell or VBScript) installed into the system that are triggered on certain conditions. WMI-based malware may even run without files on disk [47, 68].

ROP: Several researchers have looked to creating and defending against malware (particularly rootkits) built from return-oriented programming (ROP) payloads [35, 75, 108, 172].

[36] i.e., which infects systems based on traits such as keyboard layout, language, or the presence of antivirus software

7.7 Other Survey & Systematization of Knowledge Papers

Other survey and systematization of knowledge papers have covered a variety of related topics.

Egele et al. [52] survey automated dynamic malware analysis techniques and tools as of 2012, and give a brief treatment to topics in anti-analysis detection and mitigation. Neuner et al. [123] evaluate the state of the art in Android analysis systems in 2014. Pearce et al. [129] and Pék et al. [131] both survey issues in virtualization, including, but not limited to malware seeking to evade analysis in virtualized environments. Jain et al. [77] present a SoK on virtual machine introspection (VMI) and challenges associated with the technique. Ferrie's seminal technical report [56] and subsequent followup [57] describe a number of attacks that malware may use to detect popular VMMs circa 2006. Ugarte-Pedrero et al. [161] survey malware packers and their evolution.

8 CONCLUSION

We have presented a survey of evasive malware techniques, evasion detection and mitigation, and case studies on evasion. The study of evasive malware is an incredibly important and difficult topic that will continue to challenge academic and industry researchers. We hope that this survey will provide a valuable basis for future research in the field.

ACKNOWLEDGMENTS

We thank Rolf Rolles, James Kukucka, and Aaron Sedlacek for their thoughtful insights and help in reviewing and editing this paper. We also thank Jeremy Blackthorne and Greg Hughes for their support of our research. Finally, we would like to thank our anonymous reviewers for their valuable comments and suggestions for improvements - we particularly appreciate reviewer #4's insight into historical anti-analysis techniques.

REFERENCES

[1] [n. d.]. Cuckoo Sandbox. https://cuckoosandbox.org/. ([n. d.]).

[2] [n. d.]. VirtualBox. https://www.virtualbox.org. ([n. d.]).

[3] [n. d.]. VMware. http://www.vmware.com/. ([n. d.]).

[4] 2003. Norman SandBox Whitepaper. http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf. (2003).

[5] Lillian Ablon and Timothy Bogart. 2017. *Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits.* Technical Report. RAND Corporation.

[6] Alberto Ortega. [n. d.]. Paranoid Fish. https://github.com/a0rtega/pafish. ([n. d.]).

[7] Andrea Allievi and Richard Johnson. 2017. Harnessing Intel Processor Trace on Windows for Vulnerability Discovery. (2017). Talk at REcon Brussels, Brussels, Belgium.

[8] Abdulla Alwabel, Hao Shi, Genevieve Bartlette, and Jelena Mirkovic. 2014. Safe and Automated Live Malware Experimentation on Public Testbeds. In *7th Workshop on Cyber Security Experimentation and Test (CSET '14).*

[9] Various Authors. 2014. Special Issue on Biometric Spoofing and Countermeasures. *IEEE Transactions on Information Forensics and Security* 9, 6.

[10] Golam Sarwar Babil, Olivier Mehani, Roksana Boreli, and Mohamed-Ali Kaafar. 2013. On the Effectiveness of Dynamic Taint Analysis for Protecting Against Private Information Leaks on Android-based Devices. In *Proceedings of the 2013 International Conference on Security and Cryptography (SECRYPT).* IEEE.

[11] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. 2010. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *2010 29th IEEE Symposium on Reliable Distributed Systems.* IEEE.

[12] Davide Balzarotti, Marco Cova, Christoph Karlberger, and Engin Kirda. 2010. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS).*

[13] 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.

[14] Gabriel Negreira Barbosa and Rodrigo Rubira Branco. 2014. Prevalent Characteristics in Modern Malware. (2014). Talk at Black Hat 2014, Las Vegas, Nevada.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *ACM SIGOPS Operating Systems Review*, Vol. 37. ACM.

[16] Brian Bartholomew and Juan Andrés Guerrero-Saade. 2016. Wave Your False Flags! Deception Tactics Muddying Attribution in Targeted Attacks. (2016). Talk at Virus Bulletin International Conference, Denver, CO.

[17] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin. 2015. A Survey on Hypervisor-Based Monitoring: Approaches, Applications, and Evolutions. *ACM Computing Surveys (CSUR)* 48, 1 (2015).

[18] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. 2010. Improving the Efficiency of Dynamic Malware Analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing.* ACM.

[19] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. 2006. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the 15th European Institute for Computer Antivirus Research Annual Conference (EICAR 2006).*

[20] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track.*

[21] Jeremy Blackthorne, Alexei Bulazel, Andrew Fasano, Patrick Biernat, and Bülent Yener. 2016. AVLeak: Fingerprinting Antivirus Emulators Through Black-Box Testing. In *WOOT'16 Proceedings of the 10th USENIX Workshop on Offensive Technologies.* USENIX.

[22] Jeremy Blackthorne, Benjamin Fuller, Benjamin Kaiser, and Bülent Yener. 2017. Environmental Authentication in Malware. In *Latincrypt 2017.*

[23] Jeremy Blackthorne, Benjamin Kaiser, and Bülent Yener. 2016. A Formal Framework for Environmentally Sensitive Malware. In *International Symposium on Research in Attacks, Intrusions, and Defenses.* Springer.

[24] Jacob Boomgaarden, Joshua Corney, Holly Whittaker, George Dinolt, and John McEachen. 2015. Challenges in Emulating Sensor and Resource-Based State Changes for Android Malware Detection. In *2015 9th International Conference on Signal Processing and Communication Systems (ICSPCS).* IEEE.

[25] Jacob Boomgaarden, Joshua Corney, Holly Whittaker, George Dinolt, and John McEachen. 2016. Mobile Konami Codes: Analysis of Android Malware Services Utilizing Sensor and Resource-Based State Changes. In *2016 49th Hawaii International Conference on System Sciences (HICSS).* IEEE.

[26] Michael Brengel, Michael Backes, and Christian Rossow. 2016. Detecting Hardware-Assisted Virtualization. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[27] Tom Brosch and Maik Morgenstern. 2006. Runtime Packers: The Hidden Problem? (2006). Talk at Black Hat 2006, Las Vegas, NV.

[28] Gerald Brown, Matthew Carlyle, Javier Salmerón, and Kevin Wood. 2006. Defending Critical Infrastructure. *Interfaces* 36, 6 (2006).

[29] 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.

[30] Alexei Bulazel. 2016. AVLeak: Fingerprinting Antivirus Emulators For Advanced Malware Evasion. (2016). Talk at Black Hat 2016, Las Vegas, NV.

[31] Elie Bursztein, Artem Malyshey, Tadek Pietraszek, and Kurt Thomas. 2016. Picasso: Lightweight Device Class Fingerprinting for Web Clients. In *Workshop on Security and Privacy in Smartphones and Mobile Devices.*

[32] Patrick Carter, Collin Mulliner, Martina Lindorfer, William Robertson, and Engin Kirda. 2016. CuriousDroid: Automated User Interface Interaction for Android Application Analysis Sandboxes. In *Financial Cryptography and Data Security (FC).*

[33] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. 2007. *Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense.* Technical Report. Secure Systems Lab at Stony Brook University.

[34] 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.

[35] Ping Chen, Xiao Xing, Bing Mao, and Li Xie. 2010. Return-Oriented Rootkit without Returns (on the x86). In *International Conference on Information and Communications Security.* Springer.

[36] 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 *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN).* IEEE.

[37] Mihai Christodorescu and Somesh Jha. 2004. Testing Malware Detectors. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004).

[38] Zi Chu, Steven Gianvecchio, Aaron Koehl, Haining Wang, and Sushil Jajodia. 2013. Blog or block: Detecting blog bots through behavioral biometrics. *Computer Networks* 57, 3 (2013).

[39] Christian S. Collberg and Clark Thomborson. 2002. Watermarking, Tamper-Proofing, and Obfuscation-Tools For Software Protection. *IEEE Transactions on Software Engineering* 28, 8 (2002).

[40] Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2010. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the 19th International Conference on World Wide Web*. ACM.

[41] Jedidiah R Crandall, Gary Wassermann, Daniela AS de Oliveira, Zhendong Su, S Felix Wu, and Frederic T Chong. 2006. Temporal Search: Detecting Hidden Malware Timebombs with Virtual Machines. In *Conference on Architectural Support for Programming Languages and OS*.

[42] Dino Dai Zovi. 2006. Hardware Virtualization Rootkits. (2006). Talk at Black Hat 2006, Las Vegas, NV.

[43] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. SPIDER: Stealthy Binary Program Instrumentation and Debugging Via Hardware Virtualization. In *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM.

[44] Anthony Desnos, Éric Filiol, and Ivan Lefou. 2011. Detecting (and creating!) a HVM rootkit (aka BluePill-like). *Journal in Computer Virology* 7, 1 (2011).

[45] Wenrui Diao, Xiangyu Liu, Zhou Li, and Kehuan Zhang. 2016. Evading Android Runtime Analysis Through Detecting Programmed Interactions. In *WiSec '16 Proceedings of the 9th ACM Conference on Security Privacy in Wireless and Mobile Networks*.

[46] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. 2008. Ether: Malware Analysis via Hardware Virtualization Extensions. In *CCS '08 Proceedings of the 15th ACM Conference on Computer and Communications Security*.

[47] Julius Dizon, Lennard Galang, and Marvin Cruz. 2010. *Understanding WMI Malware*. Technical Report. Trend Micro.

[48] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-scale Automated Vulnerability Addition. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P 2016)*.

[49] Brendan Dolan-Gavitt and Yacin Nadji. 2010. *See No Evil: Evasions in Honeymonkey Systems*. Technical Report.

[50] Peter Eckersley. 2010. How Unique Is Your Web Browser?. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer.

[51] Robert Edmonds, Paul Royal, Mitch Halpin, Wenke Lee, and David Dagon. 2006. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. *2006 22nd Computer Security Applications Conference* (2006).

[52] Manuel Egele, Scholte Theodoor, Engin Kirda, and Christopher Kruegel. 2012. A Survey on Automated Dynamic Malware Analysis Techniques and Tools. *ACM Computing Surveys (CSUR)* 44, 2 (2012).

[53] Karim O Elish, Xiaokui Shu, Danfeng Daphne Yao, Barbara G Ryder, and Xuxian Jiang. 2015. Profiling User-Trigger Dependence for Android Security. *Computers & Security* 49 (2015).

[54] Shawn Embleton, Sherri Sparks, and Cliff C Zou. 2013. SMM Rootkits: A New Breed of OS Independent Malware. *Security and Communication Networks* 6, 12 (2013).

[55] Olivier Ferrand. 2015. How to detect the Cuckoo Sandbox and to Strengthen it? *Journal of Computer Virology and Hacking Techniques* 11, 1 (2015).

[56] Peter Ferrie. 2006. *Attacks on Virtual Machine Emulators*. Technical Report. Symantec Advanced Threat Research.

[57] Peter Ferrie. 2007. *Attacks on More Virtual Machine Emulators*. Technical Report. Symantec Advanced Threat Research.

[58] Peter Ferrie. 2016. A Brief Description of Some Popular Copy-Protection Techniques on the Apple][Platform. *PoC || GTFO 10:7* (2016).

[59] Dan Fleck, Arnur Tokhtabayev, Alex Alarif, Angelos Stavrou, and Tomas Nykodym. 2013. PyTrigger: A System to Trigger & Extract User-Activated Malware Behavior. In *Proceedings of the Eighth International Conference on Availability, Reliability and Security (ARES'13)*. IEEE.

[60] 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).

[61] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. TriggerScope: Towards Detecting Logic Bombs in Android Apps. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. San Jose, CA.

[62] Xiang Fu. 2016. On Detecting Environment Sensitivity Using Slicing. *Theoretical Computer Science* (2016).

[63] Michael N Gagnon, Stephen Taylor, and Anup K Ghosh. 2007. Software Protection through Anti-Debugging. *IEEE Security & Privacy* 5, 3 (2007).

[64] Jyoti Gajrani, Jitendra Sarswat, Meenakshi Tripathi, Vijay Laxmi, MS Gaur, and Mauro Conti. 2015. A Robust Dynamic Analysis System Preventing SandBox Detection by Android Malware. In *Proceedings of the 8th International Conference on Security of Information and Networks*. ACM.

[65] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *HOTOS'07 Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*.

[66] Steven Gianvecchio, Zhenyu Wu, Mengjun Xie, and Haining Wang. 2009. Battle of Botcraft: Fighting Bots in Online Games with Human Observational Proofs. In *CCS '09 Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM.

[67] Christian Gorecki, Felix C Freiling, Kührer Marc, and Thorsten Holz. 2011. TRUMANBOX : Improving Dynamic Malware Analysis by Emulating the Internet. In *SSS'11 Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*.

[68] Matt Graeber. 2015. Abusing Windows Management Instrumentation (WMI) to Build a Persistent, Asynchronous, and Fileless Backdoor. (2015). Talk at Black Hat 2015, Las Vegas, NV.

[69] Bernhard Grill, Andrei Bacs, Christian Platzer, and Herbert Bos. 2015. 'Nice Boots!': A Large-Scale Analysis of Bootkits and New Ways to Stop Them. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[70] Juan Andrés Guerrero-Saade. 2015. The Ethics And Perils of APT Research: An Unexpected Transition Into Intelligence Brokerage. (2015). Talk at Virus Bulletin International Conference 2015, Prague, Czech Republic.

[71] Grant Ho, Dan Boneh, Lucas Ballard, and Niels Provos. 2014. Tick Tock: Building Browser Red Pills from Timing Side Channels. In *WOOT'14 Proceedings of the 8th USENIX Workshop on Offensive Technologies*.

[72] Thorsten Holz, Markus Engelberth, and Felix Freiling. 2009. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. In *European Symposium on Research in Computer Security*. Springer.

[73] Thorsten Holz and Frederic Raynal. 2005. Detecting Honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*.

[74] Martin Hron and Jakub Jermář. 2014. SafeMachine: malware needs love, too. (2014). Talk at Virus Bulletin International Conference 2014, Seattle, WA.

[75] Ralf Hund, Thorsten Holz, and Felix C Freiling. 2009. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security '09)*.

[76] Médéric Hurier, Kevin Allix, Tegawendé François D Assise Bissyande, Jacques Klein, and Yves Le Traon. 2016. On the Lack of Consensus in Anti-Virus Decisions: Metrics and Insights on Building Ground Truths of Android Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer.

[77] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. 2014. SoK: Introspections on Trust and the Semantic Gap. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE.

[78] Christopher Jämthagen, Patrik Lantz, and Martin Hell. 2013. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. In *WATeR 2013: The First Workshop on Anti-malware Testing Research*. IEEE.

[79] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *CCS '11 Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM.

[80] Jing Jin, Jeff Offutt, Nan Zheng, Feng Mao, Aaron Koehl, and Haining Wang. 2013. Evasive Bots Masquerading as Human Beings on the Web. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE.

[81] 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.

[82] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. 2011. Differential Slicing: Identifying Causal Execution Differences for Security Applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P 2011)*. IEEE.

[83] Ryan Johnson and Angelos Stavrou. 2013. Forced-Path Execution for Android Applications on x86 Platforms. In *2013 IEEE 7th International Conference on Software Security and Reliability-Companion (SERE-C)*. IEEE.

[84] Ah Reum Kang, Jiyoung Woo, Juyong Park, and Huy Kang Kim. 2013. Online game bot detection based on party-play log analysis. *Computers & Mathematics with Applications* 65, 9 (2013).

[85] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. 2007. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*. ACM.

[86] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. 2009. Emulating Emulation-Resistant Malware. In *VMSec '09 Proceedings of the 1st ACM Workshop on Virtual Machine Security*.

[87] Alex Kantchelian, Michael Carl Tschantz, Sadia Afroz, Brad Miller, Vaishaal Shankar, Rekha Bachwani, Anthony D Joseph, and JD Tygar. 2015. Better Malware Ground Truth: Techniques for Weighting Anti-Virus Vendor Labels. In *Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security*. ACM.

[88] Alexandros Kapravelos, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2011. Escape from Monkey Island: Evading High-Interaction Honeyclients. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer.

[89] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting Malicious Behavior in Browser Extensions. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*.

[90] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. 2013. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*.

[91] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. 2016. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*. USENIX Association, Austin, TX.

[92] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-Force: Forced Execution on JavaScript. In *Proceedings of the 26th International Conference on World Wide Web*.

[93] Samuel T King and Peter M Chen. 2006. SubVirt: Implementing Malware With Virtual Machines. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE.

[94] Dhilung Kirat, Vigna Giovanni, and Christopher Kruegel. 2011. BareBox: Efficient Malware Analysis on Bare-Metal. In *ACSAC '11 Proceedings of the 27th Annual Computer Security Applications Conference*.

[95] Dhilung Kirat and Giovanni Vigna. 2015. MalGene: Automatic Extraction of Malware Analysis Evasion Signature. In *CCS '15 Proceedings of the 22nd ACM Conference on Computer and Communications Security*.

[96] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. 2014. BareCloud: Bare-metal Analysis-based Evasive Malware Detection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*.

[97] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. 2006. Behavior-based Spyware Detection. In *Proceedings of the 15th USENIX Security Symposium (USENIX Security '06)*.

[98] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. KVM: the Linux Virtual Machine Monitor. In *Proceedings of the Linux symposium*, Vol. 1.

[99] Amit Klein. 2012. Tilon: Son of Silon. https://securityintelligence.com/tilon-son-of-silon/. (2012). IBM X-Force Research.

[100] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. 2005. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing* 2, 2 (2005).

[101] Clemens Kolbitsch, Engin Kirda, and Christopher Kruegel. 2011. The Power of Procrastination: Detection and Mitigation of Execution-Stalling Malicious Code. In *CCS '11 Proceedings of the 18th ACM Conference on Computer and Communications Security*.

[102] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P 2012)*. IEEE.

[103] Christopher Kruegel, Engin Kirda, Paolo Milani Comparetti, Ulrich Bayer, and Clemens Hlauschek. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS)*.

[104] Boris Lau and Vanja Svajcer. 2008. Measuring virtual machine detection in malware using DSD tracer. *Journal in Computer Virology* 6, 3 (2008).

[105] Kevin P Lawton. 1996. Bochs: A Portable PC Emulator For Unix/X. *Linux Journal* 1996, 29 (1996).

[106] Kevin Leach, Chad Spensky, Westley Weimer, and Fengwei Zhang. 2016. Towards Transparent Introspection. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.

[107] 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.

[108] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. 2010. Defeating Return-Oriented Rootkits With "Return-Less" Kernels. In *Proceedings of the 5th European Conference on Computer Systems*. ACM.

[109] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. *arXiv preprint arXiv:1404.7431* (2014).

[110] Xiaoning Li and Kang Li. 2014. Defeating the Transparency Features of Dynamic Binary Instrumentation: The detection of DynamoRIO through introspection. (2014). Talk at Black Hat 2014, Las Vegas, NV.

[111] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. 2011. Detecting Environment-Sensitive Malware. In *RAID'11 Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*.

[112] Dominik Maier, Tilo Müller, and Mykola Protsenko. 2014. Divide-and-Conquer: Why Android Malware Cannot Be Stopped. In *Proceedings of the Ninth International Conference on Availability, Reliability and Security (ARES'14)*. IEEE.

[113] Dominik Maier, Mykola Protsenko, and Tilo Müller. 2015. A Game of Droid and Mouse: The Threat of Split-Personality Malware on Android. *Computers & Security* 54 (2015).

[114] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *Proceedings of the 19th International Symposium on Software Testing and Analysis - ISSTA '10*. ACM Press, New York, New York, USA.

[115] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis - ISSTA '09*. ACM Press, New York, New York, USA.

[116] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. 2017. Spotless Sandboxes: Evading Malware Analysis Systems using Wear-and-Tear Artifacts. In *Proceedings of the 2017 IEEE Symposium on Security and Privacy (S&P 2017)*. IEEE.

[117] Travis Morrow and Josh Pitts. 2016. Genetic Malware: Designing Payloads For Specific Targets. (2016). Talk at Infiltrate 2016, Miami, FL.

[118] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (S&P 2007)*. IEEE.

[119] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. IEEE.

[120] Alexander Moshchuk, Tanya Bragin, Steven D Gribble, and Henry M Levy. 2006. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS)*.

[121] Simone Mutti, Yanick Fratantonio, Antonio Bianchi, Luca Invernizzi, Jacopo Corbetta, Dhilung Kirat, Christopher Kruegel, and Giovanni Vigna. 2015. BareDroid: Large-Scale Analysis of Android Apps on Real Devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM.

[122] Jose Nazario. 2009. PhoneyC: A Virtual Client Honeypot. In *LEET'09 Proceedings of the 2nd USENIX Workshop On Large-Scale Exploits And Emergent Threats*, Vol. 9.

[123] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. 2014. Enter Sandbox: Android Sandbox Comparison. *arXiv preprint arXiv:1410.7749* (2014).

[124] 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 *2009 Annual Computer Security Applications Conference*. IEEE.

[125] Jon Oberheide. 2008. Detecting and Evading CWSandbox. https://jon.oberheide.org/blog/2008/01/15/detecting-and-evading-cwsandbox/. (2008).

[126] Jon Oberheide and Charlie Miller. 2012. Dissecting The Android Bouncer. (2012). Talk at SummerCon 2012, Brooklyn, New York.

[127] Tavis Ormandy. 2007. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. (2007). Talk at CanSecWest Applied Security Conference, Vancouver, CA.

[128] 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 *WOOT'09 Proceedings of the 3rd USENIX Workshop on Offensive Technologies*.

[129] Michael Pearce, Sherali Zeadally, and Ray Hunt. 2013. Virtualization: Issues, Security Threats, and Solutions. *ACM Computing Surveys (CSUR)* 45, 2 (2013).

[130] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. 2011. nEther: In-guest Detection of Out-of-the-guest Malware Analyzers. In *EUROSEC '11 Proceedings of the Fourth European Workshop on System Security*.

[131] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. 2013. A Survey of Security Issues in Hardware Virtualization. *ACM Computing Surveys (CSUR)* 45, 3 (2013).

[132] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*.

[133] Nicholas Percoco and Sean Schulte. 2012. Adventures in BouncerLand: Failures of Automated Malware Detection within Mobile Application Markets. (2012). Talk at Black Hat 2012, Las Vegas, NV.

[134] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware. In *EUROSEC '14 Proceedings of the Seventh European Workshop on System Security*. ACM.

[135] Mario Polino, Andrea Continella, Sebastiano Mariani, Stefano D'Alessio, Lorenzo Fontana, Fabio Gritti, and Stefano Zanero. 2017. Measuring and Defeating Anti-Instrumentation-Equipped Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer.

[136] Michalis Polychronakis, Panayiotis Mavrommatis, and Niels Provos. 2008. Ghost Turns Zombie: Exploring the Life Cycle of Web-based Malware. In *LEET'08 Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats*. USENIX Association, Berkeley, CA, USA.

[137] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. 2007. Detecting System Emulators. In *International Conference on Information Security*. Springer.

[138] Moheeb Abu Rajab, Lucas Ballard, Nav Jagpal, Panayiotis Mavrommatis, Daisuke Nojiri, Niels Provos, and Ludwig Schmidt. 2011. *Trends in Circumventing Web-Malware Detection.* Technical Report. Google.

[139] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. 2016. Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS).*

[140] Lars Richter. 2015. *Common Weaknesses of Android Malware Analysis Frameworks.* Technical Report. University of Erlangen-Nuremberg.

[141] Phillip Rogaway. 2015. The Moral Character of Cryptographic Work. Cryptology ePrint Archive, Report 2015/1162. (2015). http://eprint.iacr.org/2015/1162.

[142] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *WOOT'09 Proceedings of the 3rd USENIX Workshop on Offensive Technologies.*

[143] Rolf Rolles. 2015. Memory Lane: Hacking Renovo. http://www.msreverseengineering.com/blog/2015/7/16/hacking-renovo. (2015).

[144] Lee M Rossey, Robert K Cunningham, David J Fried, Jesse C Rabek, Richard P Lippmann, Joshua W Haines, and Marc A Zissman. 2002. LARIAT: Lincoln Adaptable Real-time. Information Assurance Testbed. In *Aerospace Conference Proceedings, 2002. IEEE*, Vol. 6. IEEE.

[145] Christian Rossow, Christian J Dietrich, Chris Grier, Christian Kreibich, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten Van Steen. 2012. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (S&P 2012).* IEEE.

[146] Kevin A Roundy and Barton P Miller. 2013. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys (CSUR)* 46, 1 (2013).

[147] Paul Royal. 2012. Entrapment: Tricking Malware with Transparent, Scalable Malware Analysis. (2012). Talk at Black Hat 2012, Las Vegas, NV.

[148] Joanna Rutkowska. 2004. Red Pill… Or How To Detect VMM Using (Almost) One CPU Instruction. http://www.securiteam.com/securityreviews/6Z00H20BQS.html. (2004).

[149] Joanna Rutkowska. 2006. Introducing Blue Pill. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html. (2006).

[150] Joshua Schiffman and David Kaplan. 2014. The SMM Rootkit Revisited: Fun with USB. In *Proceedings of the Ninth International Conference on Availability, Reliability and Security (ARES'14).* IEEE.

[151] V. Selis and A. Marshall. 2015. MEDA: A Machine Emulation Detection Algorithm. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, Vol. 04.

[152] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS).*

[153] Tyler Shields. 2010. *Anti-Debugging - A Developers View.* Technical Report.

[154] Chengyu Song and Paul Royal. 2012. Flowers for Automated Malware Analysis. (2012). Talk at Black Hat 2012, Las Vegas, NV.

[155] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, and Pongsin Poosankam. 2008. BitBlaze : A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security (ICISS).*

[156] Chad Spensky, Hongyi Hu, and Kevin Leach. 2016. LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS).*

[157] Guillermo Suarez-Tangil, Mauro Conti, Juan E Tapiador, and Pedro Peris-Lopez. 2014. Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models. In *European Symposium on Research in Computer Security.* Springer.

[158] Ke Sun, Xiaoning Li, and Ya Ou. 2016. Break Out of The Truman Show: Active Detection and Escape of Dynamic Binary Instrumentation. (2016). Talk at Black Hat Asia 2016, Singapore, Singapore.

[159] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS).*

[160] Christopher Thompson, Maria Huntley, and Chad Link. 2010. *Virtualization Detection: New Strategies and Their Effectiveness.* Technical Report. University of Minnesota.

[161] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P 2015).* IEEE.

[162] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo Garcia Bringas. 2016. RAMBO: Run-Time Packer Analysis with Multiple Branch Observation. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).* Springer.

[163] Phani Vadrevu and Roberto Perdisci. 2016. MAXS: Scaling Malware Execution with Sequential Multi-Hypothesis Testing. In *ASIA CCS '16 Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security.* ACM.

[164] Amit Vasudevan. 2007. *WiLDCAT: An Integrated Stealth Environment For Dynamic Malware Analysis.* Ph.D. Dissertation. University of Texas at Arlington.

[165] Amit Vasudevan and Ramesh Yerraballi. 2004. SAKTHI: A Retargetable Dynamic Framework for Binary Instrumentation. In *Hawaii International Conference in Computer Sciences.*

[166] Amit Vasudevan and Ramesh Yerraballi. 2005. Stealth Breakpoints. In *21st Annual Computer Security Applications Conference (ACSAC'05).* IEEE.

[167] Amit Vasudevan and Ramesh Yerraballi. 2006. Cobra: Fine-grained Malware Analysis using Stealth Localized-executions. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P 2006)*, Vol. 9.

[168] Amit Vasudevan and Ramesh Yerraballi. 2006. SPiKE: Engineering Malware Analysis Tools using Unobtrusive Binary-Instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference.* Australian Computer Society, Inc.

[169] Timothy Vidas and Nicolas Christin. 2014. Evading Android Runtime Analysis via Sandbox Detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security.* ACM.

[170] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. 2014. A5: Automated Analysis of Adversarial Android Applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices (SPSM '14).* ACM.

[171] Giovanni Vigna. 2014. Now You See Me, Now You Don't: Chasing Evasive Malware. https://www.slideshare.net/lastlinesecurity/ip-expo-oct-2014. (2014).

[172] Sebastian Vogl, Jonas Pfoh, Thomas Kittel, and Claudia Eckert. 2014. Persistent Data-only Malware: Function Hooks without Code. In *Proceedings of the Symposium on Network And Distributed System Security (NDSS).*

[173] Heinrich Von Stackelberg. 1952. *The Theory of The Market Economy.* Oxford University Press.

[174] Yi-Min Wang, Doug Beck, Xuxian Jiang, Roussi Roussev, Chad Verbowski, Shuo Chen, and Sam King. 2006. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS).*

[175] Ollie Whitehouse. 2014. Introduction to Anti-Fuzzing: A Defence in Depth Aid. (2014). https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2014/january/introduction-to-anti-fuzzing-a-defence-in-depth-aid/

[176] Andre Wichmann and Elmar Gerhards-Padilla. 2012. Using Infection Markers as a Vaccine against Malware Attacks. In *2012 IEEE International Conference on Green Computing and Communications (GreenCom).* IEEE.

[177] Carsten Willems, Thorsten Holz, and Felix Freiling. 2007. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy Magazine* 5, 2 (2007).

[178] 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.

[179] Michelle Y. Wong and David Lie. 2016. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS).*

[180] Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2016. Comprehensive Analysis and Detection of Flash-based Malware. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[181] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2013. AUTOVAC: Automatically Extracting System Resource Constraints and Generating Vaccines for Malware Immunization. In *2013 IEEE 33rd International Conference on Distributed Computing Systems (ICDCS).* IEEE.

[182] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. 2014. GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment. In *RAID'14 Proceedings of the 17th International Symposium on Recent Advances in Intrusion Detection.*

[183] Lok-kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. 2012. V2E : Combining Hardware Virtualization and Software Emulation for Transparent and Extensible Malware Analysis. In *VEE '12 Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments.*

[184] Wei Yan, Zheng Zhang, and Nirwan Ansari. 2008. Revealing Packed Malware. *IEEE Security & Privacy* 6, 5 (2008).

[185] Heng Yin and Dawn Song. 2010. *TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution.* Technical Report. EECS Department, University of California, Berkeley.

[186] Akira Yokoyama, Kou Ishii, Rui Tanabe, Yinmin Papa, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, Michael Brengel, Michael Backes, and Christian Rossow. 2016. SandPrint: Fingerprinting Malware Sandboxes to Provide Intelligence for Sandbox Evasion. In *RAID'16 Proceedings of the 19th International Symposium on Recent Advances in Intrusion Detection.* ACM.

[187] 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).

[188] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications.*

[189] Adam Young and Moti Yung. 2004. *Malicious Cryptography: Exposing Cryptovirology*. John Wiley & Sons.

[190] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. 2015. Using Hardware Features for Increased Debugging Transparency. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P 2015)*. IEEE.

[191] 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)*. IEEE.

[192] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. 2012. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM.