Dynamic Malware Analysis in the Modern Era—A State of the Art Survey

ORI OR-MEIR, NIR NISSIM, YUVAL ELOVICI, and LIOR ROKACH, Ben-Gurion University of the Negev, Beer-Sheva, Israel

Although malicious software (malware) has been around since the early days of computers, the sophistication and innovation of malware has increased over the years. In particular, the latest crop of ransomware has drawn attention to the dangers of malicious software, which can cause harm to private users as well as corporations, public services (hospitals and transportation systems), governments, and security institutions. To protect these institutions and the public from malware attacks, malicious activity must be detected as early as possible, preferably before it conducts its harmful acts. However, it is not always easy to know what to look for—especially when dealing with new and unknown malware that has never been seen. Analyzing a suspicious file by static or dynamic analysis methods can provide relevant and valuable information regarding a file's impact on the hosting system and help determine whether the file is malicious or not, based on the method's predefined rules. While various techniques (e.g., code obfuscation, dynamic code loading, encryption, and packing) can be used by malware writers to evade static analysis (including signature-based antivirus tools), dynamic analysis is robust to these techniques and can provide greater understanding regarding the analyzed file and consequently can lead to better detection capabilities. Although dynamic analysis is more robust than static analysis, existing dynamic analysis tools and techniques are imperfect, and there is no single tool that can cover all aspects of malware behavior. The most recent comprehensive survey performed in this area was published in 2012. Since that time, the computing environment has changed dramatically with new types of malware (ransomware, cryptominers), new analysis methods (volatile memory forensics, side-channel analysis), new computing environments (cloud computing, IoT devices), new machine-learning algorithms, and more. The goal of this survey is to provide a comprehensive and up-to-date overview of existing methods used to dynamically analyze malware, which includes a description of each method, its strengths and weaknesses, and its resilience against malware evasion techniques. In addition, we include an overview of prominent studies presenting the usage of machine-learning methods to enhance dynamic malware analysis capabilities aimed at detection, classification, and categorization.

CCS Concepts: • **Security and privacy** → *Intrusion/anomaly detection and malware mitigation*; *Malware and its mitigation*; *Intrusion detection systems*;

Additional Key Words and Phrases: Dynamic analysis, malware, detection, evasion, behavioral analysis

ACM Reference format:
Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* 52, 5, Article 88 (September 2019), 48 pages.
https://doi.org/10.1145/3329786

Authors' addresses: O. Or-Meir, Y. Elovici, and L. Rokach, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel; Department of Software and Information Systems Engineering, Ben-Gurion University of the Negev, Beer-Sheva, Israel; emails: oriormeir@gmail.com, elovici@bgu.ac.il, liorrk@post.bgu.ac.il; N. Nissim, Malware Lab, Cyber Security Research Center, Ben-Gurion University of the Negev, Beer-Sheva, Israel; Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva, Israel; email: nirni@bgu.ac.il.

0360-0300/2019/09-ART88 $15.00
https://doi.org/10.1145/3329786

1 INTRODUCTION

In recent years, malware attacks have increased dramatically. The costs associated with ransomware damage are expected to reach $8 billion in 2018.[1] Although 94% of the businesses that have suffered a ransomware attack did not pay the ransom, the harm caused by data loss and the number of man-hours required to mitigate the infection can be significant. After peaking in 2016, ransomware attacks have become less common.[2] In contrast, cryptomining, which utilizes the victim's computing power to mine cryptocurrencies for the benefit of the attacker, has increased.[3]

To counter malicious attacks on computing systems there is a need to detect malware as early as possible and prevent it from executing its malicious code. While it is usually easy to detect known malware, the main problem is handling unknown binary code. To determine whether an unknown/new executable is malicious or not, it is common to use an expert analyst who can assess the nature of the executable. If the expert's analysis reveals that the executable is malicious, then a signature pattern (based on static, dynamic, or hybrid features) can be crafted to provide detection tools with the capability of detecting the attack in the future (including similar variants). In other words, analysis turns "never seen before malware" into a signature that can be detected in the future. While manual analysis is very reliable, it is not scalable, nor is it possible to apply such analysis on every file due to the heavy costs associated with manual analysis.

To overcome this problem, automatic tools for analyzing unknown executables have been developed. Automatic analysis tools employ different techniques to track the behavior of the sample being analyzed and produce a report that describes the different actions taken by the executable; for most of these tools, the report includes heuristic classification regarding the nature of the sample (malicious or benign).

Examination of unknown executables can be performed by analyzing the code statically or dynamically. Static analysis is based on the extraction of information that provides hints about the code's behavior; dynamic analysis is based on the idea that you can execute the code and actually trace what it does and how it affects the hosting system.

While no tool can be perfect, automatically analyzing code can significantly reduce the workload for the human analyst. By classifying and filtering out files that can be detected with known techniques, the human analyst only needs to handle the never seen before samples of code, which are few, given the fact that malware authors usually reuse malicious code.

Analysis techniques have evolved dramatically over the years, and they are now capable of detecting more and more attack mechanisms and evasion methods. However, malware authors have been developing new techniques of their own to evade or subvert the analysis tools and cause them to misclassify a malicious file as benign. Malware typically employs as many as 10 evasion techniques per sample,[4] which indicates both that malware analysis is a great concern of malware authors and that they are aware of the efforts taken to develop effective malware detection methods.

Dynamic analysis [1] refers to the process of analyzing a code or script by executing it and observing its actions. These actions can be observed at various levels, from the lowest level possible (the binary code itself) to the system as a whole (e.g., changes made to the registry or file system). The objective of dynamic analysis is to expose the malicious activity performed by the executable while it is running, without compromising the security of the analysis platform. From the defensive

[1] https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-exceed-8-billion-in-2018/.
[2] https://cdn.sonicwall.com/sonicwall.com/media/pdfs/resources/2018-snwl-cyber-threat-report.pdf.
[3] https://www.symantec.com/content/dam/symantec/docs/reports/istr-23-2018-en.pdf.
[4] https://www.lastline.com/labsblog/labs-report-at-rsa-evasive-malwares-gone-mainstream/.

perspective, there is a risk of being infected by the malware while analyzing it dynamically, since it requires that the malware be loaded into the RAM and executed by the hosting CPU.

Dynamic analysis, unlike static analysis, does not rely on analyzing the binary code itself and looks for meaningful patterns or signatures that imply maliciousness of the analyzed file. Such a static approach is vulnerable to many evasion techniques (e.g., packing, obfuscation, etc.). In addition, dynamic analysis does not translate the binary code to assembly level code (AKA disassembling). While a disassembling process may seem straightforward, there are several techniques attackers use to fool the disassembler program and cause it to produce a different assembly code than the one that is actually being executed. By avoiding the disassembling process and by not relying on the analyzed file's binary code itself, dynamic analysis is immune to such malware evasion techniques (for other evasion techniques, see Section 4.2). In addition, while static analysis cannot detect changes made to the code during execution, dynamic analysis is immune to this effect.

The last comprehensive survey in the domain of dynamic malware analysis was conducted in 2012. Since then, new malware types (such as ransomware and cryptominers), as well as new analysis techniques (such as volatile memory forensics and side-channel analysis), have emerged. This survey was conducted to address this gap and provide researchers with important information on advancements in this field.

Although surveys in the domain of malware analysis have been performed in the past, they are either out of date or narrow in scope (e.g., focused on the difference between static and dynamic analysis [2–4]). Another paper surveyed and presented various analysis techniques without going into details [5]; one survey provides a comparison between surveyed tools [6]; another focused on malware propagation methods [7]; and two surveys compared different machine-learning and data-mining studies conducted on malware analysis [8, 9]. In recent years, there has been a large increase in the number of published papers regarding machine-learning-based solutions and the different domains in which they can be applied. Thus, there is no surprise that some studies showed that unknown malware detection can be improved using machine-learning algorithms that provide prediction, generalization, and classification capabilities. The abovementioned surveys may be useful to researchers interested in increasing their knowledge regarding some of the analysis techniques developed in the past, including the core techniques used to dynamically collect relevant data. Such knowledge can be leveraged to enhance malware detection capabilities.

Since the survey performed by Egele et al. in 2012 [6] has been published, no survey has provided a comprehensive comparison between dynamic analysis methods for malware detection; this is problematic for the following reasons:

(1) The use of embedded devices and technologies, including attractive malware targets like IoT devices, wearable devices, digital medical devices, and autonomous vehicles, has become widespread since 2012, however analysis techniques developed at the time of Egele's survey were not designed to analyze malicious code on such embedded systems and don't have the necessary detection capabilities.
(2) In 2011, side-channel analysis [10], a **novel and significant** dynamic malware analysis technique, was developed based on an idea presented in 1997 [11]. This technique relies on analyzing malware by using *hardware performance counters* (HPCs) and other hardware indications. Since 2012, five papers have been published [12–16] providing additional analysis capabilities based on the original side-channel idea. This method added generic and cross platform analysis capabilities for malware detection. However, the studies dealing with this technique have not yet been surveyed and compared.
(3) The increasingly popular cryptocurrency (e.g., Bitcoin, Monero) trend and the anonymity such currency provides, have led to the development and popularity of a new generation

of *ransomware* (although the first ransomware was developed in 1989,[5] ransomware became wide spread in 2013 with the increase use of cryptocurrency). In addition, this trend resulted in the creation and rise of a new type of malware called cryptominer, in which the malware hijacks and utilizes the victim's computational resources to significantly increase the attacker's mining capacity and monetary profit. To the best of our knowledge, no survey exists that discusses these malware types in terms of their analysis or detection.

(4) In 2011, Vomel et al. [17] published a survey about volatile memory forensic, however this technique was not covered in the 2012 survey [6]. Other surveys published since 2012 were not comprehensive enough, failing to provide a comparison between volatile memory forensics and other techniques, although such comparisons are always needed, especially in cloud computing ecosystems.

Web and cloud servers are an attractive target for cyber-attackers, because they contain sensitive data that can be stolen, are critical to organizations, and have strong CPUs that attackers can use for cryptomining. Since servers are rarely shut down and must be available at all time, the analysis of such machines has been very limited. For this reason, volatile memory forensic can play a role in the analysis of such systems. IoT devices are another example of a platform that can benefit from memory analysis, as the design and installation of new software-based detection mechanisms is not straightforward. In addition, because IoT devices have limited computational resources, existing dynamic analysis techniques might be less relevant or effective for such devices.

(5) Surveys on machine-learning methods for malware detection [9] and malware detection using dynamic analysis [8] have been presented, however these surveys don't provide the reader with comprehensive information or a thorough analysis regarding the machine-learning methods that leverage dynamic analysis techniques for the task of malware detection and categorization. Today, such information is vital for the data science and cyber security scientific communities, since it will enable readers to easily understand which combination of machine-learning methods and dynamic analysis techniques provide better detection capabilities. This information is relevant, particularly in light of the contribution of machine-learning methods in a variety of domains and the sheer volume of new malware created daily. Such a reality requires the generalization of detection capabilities that can be provided by machine-learning algorithms [18–21].

In addition to our contributions with regard to the five issues mentioned above, we also present three new taxonomies: classification of malware based on the behavior presented (Section 3.2), classification of malware based on the privilege it operates in (Section 3.3), and a taxonomy of malware behavior (Section 4). These taxonomies help anticipate the behavior a malware can present, design the analysis process of recording each operation and defending against a malicious operation.

We also include a comparison between analysis techniques based on the malware behavior(s) they subvert or analyze (Section 9). This comparison can be useful to researchers interested in designing analysis tools by helping them consider various scenarios when analyzing live malware. We believe this comparison will help create more secure and trusted analysis frameworks in the future.

In addition, we build on the work performed by Egele et al. [6], providing a comprehensive and up-to-date comparison of academic studies of dynamic analysis methods, including studies published both in recent years and prior to 2012 (Section 10).

[5] https://digitalguardian.com/blog/history-ransomware-attacks-biggest-and-worst-ransomware-attacks-all-time.

Finally, we wish to emphasize the scope of our study. Malware dynamic analysis is a broad domain that is difficult to cover in a single survey. For brevity, we present the various analysis techniques published since 2004, and we include several examples of academic tools for each technique (Section 7). Presenting a full list of the academic tools would result in an unwieldly survey, containing dynamic analysis studies that either repeat one another or have had little impact. Thus, we focused on significant academic papers that presented an analysis tool for the Windows Operating System (OS), which is the most widely used OS. In terms of the analysis of IoT and medical devices, we present those papers that addressed *side-channel analysis*. This new dynamic analysis technique does not query the operating system to perform the analysis, and thus it can be applied to various OSs or IoT devices (see Sections 6.6 and 7.5).

2 WHAT IS MALWARE?

"Malware" is an acronym for **mal**icious soft**ware**, which refers to any script or binary code that performs some malicious activity. Malware can come in different formats, such as executables, binary shell code, script, and firmware. In this survey, when referring to malware, we also use the term "malicious binary code," but the terms "malicious script" or "malicious executable" are also acceptable. Before presenting our definitions, we review a few prior definitions of malware:

2.1 Previous Definitions

- *A program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data, applications, or operating system or otherwise annoying or disrupting the victim.* **—NIST**[6]
- *Any program or file that is harmful to a computer user. Malicious programs can perform a variety of functions, including stealing, encrypting or deleting sensitive data, altering or hijacking core computing functions and monitoring users' computer activity without their permission.* **—TechTarget**[7]
- *A computer program designed to infiltrate and damage computers without the user's consent.* **—BullGuard**[8]
- *A type of computer program designed to infect a legitimate user's computer and inflict harm on it in multiple ways. Malware can infect computers and devices in several ways and comes in a number of forms, just a few of which include viruses, worms, Trojans, spyware and more.* **—Kaspersky**[9]
- *Malware is software that is specifically designed to gain access or damage a computer without the knowledge of the owner.* **—Norton**[10]

These definitions are very similar to one another, and they all include at least one of the following two main traits: (1) maliciousness or harmfulness and (2) the ability to perform actions without the user's consent or knowledge.

2.2 Our Definition

Previous definitions of malware largely focus on the intentions of malware authors, however such definitions are inadequate, since it is impossible to determine the intentions behind an unknown binary code.

[6] https://csrc.nist.gov/Glossary/?term=5373.
[7] https://searchsecurity.techtarget.com/definition/malware.
[8] https://www.bullguard.com/bullguard-security-center/pc-security/computer-threats/malware-definition,-history-and-classification.aspx.
[9] https://www.kaspersky.com/resource-center/preemptive-safety/what-is-malware-and-how-to-protect-against-it.
[10] https://us.norton.com/internetsecurity-malware.html.

Therefore, we present our definition for the term malware, which we believe encompasses the malicious aspects of malware:

> ***Malware is code running on a computerized system whose presence or behavior the system administrators are unaware of; were the system administrators aware of the code and its behavior, they would not permit it to run.***
>
> ***Malware compromises the confidentiality, integrity or the availability of the system by exploiting existing vulnerabilities in a system or by creating new ones.***

The first part of our definition refers malware from the system administrators' point of view. It is imperative that all code running in a system will be known and authorized by the administrator. A few examples of malicious behavior commonly found in malware: stealing sensitive information, stealing computing resources, denying service to critical components of the system, or simply annoying the infected user. We explain more about malicious behavior in Section 3.2.

The second part of our definition refers to three principals of security: ***confidentiality, integrity***, and ***availability*** (known as the *CIA triad*).[11] When a user accesses a computerized service or system, he/she assumes that it is safe in those three respects. In other words, the system is believed to be available at all times, the system and the data it contains is correct and complete, and the private information stored on or used by the system/service is not exposed to unprivileged parties. Each of the examples of malicious behavior mentioned above compromises the trust between the user and the attacked system in one (or more) of these respects (see Section 3.2).

A *vulnerability* is an error, bug, or mistake in the system that can by exploited by a third party (an attacker), to compromise one (or more) of the *CIA triad*. Exploiting a vulnerability allows the attacker to manipulate information or behavior in a way that allows them to use legitimate resources to perform malicious activities. For example, vulnerabilities such as buffer overflows and validation errors are caused by input mishandling. A system is only secure if it never trusts incoming input and always validates its content and origin. Assuming that a user always provides correct and complete data is a risky practice that often results in the creation of vulnerabilities. Exploiting such vulnerabilities is possible by sending a specially crafted input that can overrun an unprotected buffer or cause an out of bound exception.

If an attacker discovers a vulnerability in a targeted system, then it is very likely that he/she will exploit it. The vulnerability can be publicly known or a new vulnerability that has never been seen before (also known as a *zero-day* vulnerability) [22].

We believe that the importance of our definition lies in the fact that it does not refer to the intentions of the software authors, which are unknown (for example, definitions like "software developed with intent to cause damage to its target"). Instead, our definition focuses on the system owner/administrator, and his/her knowledge of the (potential) malware's behavior, which is always well defined. In addition, our definition clearly draws a line between "buggy" behavior (such as spreading software updates or security software that clogs the network by mistake) and "bad" behavior (such as spreading worms that clog the network). In the first case, the code is known to the administrator and the "malicious" effect is merely a bug, while in the second case it is malicious (mainly because it was not authorized by the administrator). Both cases have the same outcome but are totally different in nature.

[11] https://resources.infosecinstitute.com/cia-triad/.

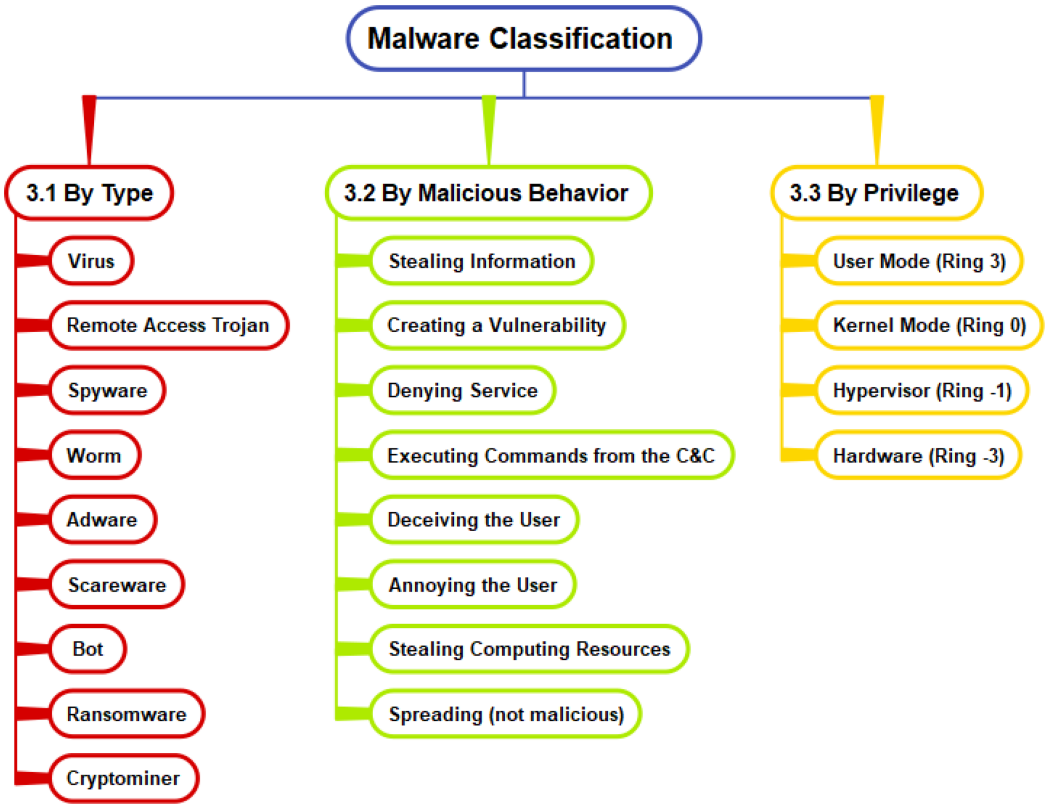

Fig. 1. Various malware taxonomies.

3 MALWARE CLASSIFICATION

Several distinctions between malware can be made, depending on the aspect being considered. The classic malware classification is based on common names given to malware and are frequently used by the average person with no prior knowledge about malware analysis.

In Figure 1, we present three malware classifications: (1) traditional classification by type, (2) classification by the behavior of the code, and (3) the privileges the malware executes.

3.1 Classification of Malware by Type

The following classification by malware type can be found in most of the literature; this terminology is also widely known and used by the general public.

A ***virus*** injects its malicious code into other files, thus spreading within the host (and potentially to other hosts as well). The term ***virus*** is often used by the mass media to describe any kind of malware.

A ***remote access Trojan*** (RAT, or *Trojan horse*) is a type of malicious software that pretends to be harmless, like the mythological wooden horse that the Greeks sent to Troy.[12] Most of the time, ***RAT***s apply social engineering techniques and hide from the user by disguising themselves as benign software, a useful tool, or game, to stay in the background and perform their malicious tasks.

[12] https://en.wikipedia.org/wiki/Trojan_Horse.

Spyware tracks the user without his/her consent and reports back to the attacker about the user's activities, visited websites, geographic location, and so on.

A ***worm*** is a program that duplicates itself and spreads through networks. Worms can spread very quickly and disrupt system use by clogging the network. Once the ***worm*** is detected, it is often easy to patch the system and prevent it from spreading further.

Adware automatically shows advertisements to the user. These ads can be injected into other software or Web pages, and in some cases, ***adware*** can even replace an existing ad with another ad. Advertising space is sold by the ***adware***'s author to vendors and companies, generating profits for the author. This type of malware does not usually harm the system, and most of the times the user will never be able to tell that he/she was infected; for this reason, ***adware*** is also referred to as ***grayware***.

Scareware presents an interface that informs users they have been infected with a malware. The interface is designed to look like an anti-virus and includes an offer to purchase software to remove the malware. After the victim purchases and installs the software, the ***scareware*** is removed by the purchased software. While most ***scareware*** does not harm the infected machine in any way, it deceives the user with false detection messages and can lead the user to believe that an actual anti-virus tool was installed.

A ***Bot*** (derived from the word robot) is a malware that performs actions without the user's consent as part of an army of "zombies." Such actions include visiting websites, spreading the malware to other hosts, and querying a service (such as DNS or email servers). All of the ***bots*** receive their instructions from a command and control (C&C) server, and the group of infected hosts is referred to as a *botnet. Botnets* are mainly used for *distributed denial-of-service* (DDoS) attacks (see Section 3.2), in which many infected devices (sometimes millions) are used simultaneously to overload a Web service (websites, DNS, cloud services, etc.). By creating a swarm of requests, the target server is flooded and unable to provide service for legitimate users.

With increased interest in cryptocurrencies, their rising value, and the inherent anonymity associated with cryptographic currency, two new types of malware have been developed to profit malicious attackers while keeping their identities secret:

Ransomware encrypts the user's files (documents and photos) with a strong form of encryption and demands payment in exchange for the decryption key. Usually, this type of behavior does not prevent using the computer, but it effectively renders all of the information on the computer inaccessible. Users who failed to back up their files before the attack and its file encryption are left helpless and forced to choose between paying a high ransom or giving up completely, formatting the entire host and losing everything. To make things worse, most ***ransomware*** demands payment within a short timeframe (usually a few days). Modern ***ransomware*** demands payment in Bitcoin or other cryptocurrencies, because they allow the attacker to remain relatively anonymous.

Cryptominers use cryptocurrency differently than *ransomware*. Instead of forcing infected users to pay a ransom to the attacker, the malware uses any available computing power of the victim to mine cryptocurrencies for the attacker. The victims bear the brunt of the increased electricity costs and performance deterioration, while the attacker makes all the profit. This is a fairly new attack that has been on the rise since 2017; reportedly, almost 90% of recent malware attacks are ***cryptominers***.[13] This trend is explained by the fact that mining cryptocurrencies yields great profits to attackers while requiring minimal effort.

[13] https://www.imperva.com/blog/2018/02/new-research-crypto-mining-drives-almost-90-remote-code-execution-attacks/.

3.2 Classification of Malware by Malicious Behavior

The terms mentioned in the previous section are frequently used in the mass media and among users to describe different types of malware. When describing malware to non-professionals, the categories and terms mentioned above are acceptable. However, for analysis purposes, it is more important to focus on malware behavior instead of malware type. It should be noted after analyzing binary code and obtaining information about its behavior, it is much easier to classify it to the correct malware type.

Stealing information—The most common malicious activity is information theft, whether it's banking information, classified information, passwords, or access credentials. Such information can be used to access other systems or accounts, steal money or other goods, or damage a company financially, and when it comes to hospitals and governmental agencies, it can even risk lives. In terms of the *CIA triad*, information theft compromises the *confidentiality* of data and is mostly associated with *RAT*s and *spyware*.

Information theft violates individuals' privacy and cause organizations to lose a large amount of money and jeopardize their reputation, particularly after the establishment of the General Data Protection Regulation (GDPR) [23], which imposes significant fines on vendors whose customers have suffered from privacy violation. Toch et al. [24] recently presented a comprehensive survey of common and novel cyber-attacks and analyzed them according to the type of privacy invasion. They also suggested a taxonomy for the assessment of privacy risks of information security technologies.

There are several examples of how information is stolen by malware, including:

- ***Stealing Credentials***—Passwords, as well as other credential types, such as biometric information and authentication certificates, are obvious targets for theft by malware.
- ***Keylogging***—A keylogger malware records every keystroke entered by the user and sends it to its C&C. Keyloggers are primarily used to extract passwords and other types of sensitive information.
- ***Spying***—*Spyware* collects information about the infected host, users, or the organization. Such information may include proprietary information, data about clients, or personal documents. The attacker might share the information with a third party (like advertisers) and make a profit.
- ***Sniffing attack***—Sniffer malware steal information as it travels from one machine to another. This is usually done by monitoring the network traffic and extracting data from it.

Creating a vulnerability—As explained in Section 2.2, infecting a host with malware is done by exploiting a vulnerability. However, malware sometimes create new vulnerabilities by removing anti-viruses, installing backdoors, changing passwords, changing firewall settings, adding a new privileged user, downgrading software to an older version,[14] and more. These new vulnerabilities can later be used by the attacker to gain access to the victim, even if the original malware was detected and removed. This behavior compromises the *integrity* of the system and associated with *RAT*s and *bots*.

Denying service—When services are constantly used, as is the case today, a denial-of-service can be hazardous as it compromises the *availability* of services. Attackers can deny service in different ways:

- ***DDoS (distributed denial-of-service) attacks***—The basic idea behind a *DDoS* attack is to generate an overwhelming number of fake requests to the same service, thus preventing

[14] https://blog.trendmicro.com/backdoor-attacks-work-protect/.

legitimate users from receiving the service they desire. *DDoS* attacks use *botnets* to generate millions (and sometimes billions) of simultaneous requests from all over the globe. A successful *DDoS* attack is not easy to defend against or trace, because each bot may send just a few requests, and it is hard to differentiate these requests from a request from a legitimate user who requires service. For the same reason it is difficult to trace the origin of the attack. On the plus side, *DDoS* attacks don't last forever. Unless the system is compromised or infected, the service will run smoothly again once the requests cease.

- ***Access denial attacks***—*Worms* can spread and cause a denial-of-service attack against the network's infrastructure (we explain more about spreading later in this section). *Ransomware* encrypts all documents and pictures on the infected host causing a denial-of-service attack against the user's file system.
- ***Damaging the hardware***—Although this behavior is not common, it is real. Since electronic circuits often contain firmware, it is possible to overwrite it with garbage code and render the device unusable. Such an attack against the BIOS was demonstrated by *Win9x.CIH* (aka *Chernobyl*), which prevents the computer from booting after overwriting the BIOS with buggy firmware code.[15]

Executing commands from the C&C—Sometimes malware authors wish to have full control over the infected endpoint. For that purpose, they incorporate a command execution mechanism in the malware that executes commands it receives from the C&C. This behavior compromises the *integrity* of the system and is mostly associated with *bots* and *RAT*s.

Deceiving the user—Malicious parties can use deception to gain access to restricted subsystems and/or manipulate data for their benefit (using stolen credentials, presenting fake messages, social engineering, etc.). This type of behavior is common among financial malware performing phishing attacks to gain access to banking accounts and transfer money from the victim's account to the attacker. This behavior compromises the *integrity* and *confidentiality* of the system and associated with *RAT*s and *scareware*. Common deception behaviors include the following: disguising as a benign software, *phishing* attacks, pretending to be a legitimate user, scaring the user, and *man-in-the-middle* (MITM) attacks.

Annoying the user—Malware might annoy the user by showing ads to the user (*adware*), changing the user's homepage or default search engine, or defacing Web pages. This type of malware behavior might not be malicious per se, but it still falls within the definition proposed in Section 2.2 in that it is unwanted by the user, and if the system administrators knew about its existence, they would not allow it to execute. Although this behavior does not compromise any component of the *CIA triad* it is unwanted by the user and disturbs normal workflow.

Stealing computing resources—*Cryptominers* leverage their victims' computer resources, causing the infected machines to run abnormally slow, for the sole purpose of generating revenue by selling the mined cryptocurrency. For the most part, *cryptominers* allow the system to run as normal and don't interact with the system's information, apart from performing the computations required to mine cryptocurrencies. This behavior compromises the *integrity* and *availability* of the system.

Spreading—A common behavior seen in *worms* and *viruses* is spreading. An example where spreading is malicious is the *Morris worm*, which spread by exploiting a vulnerability in the BSD operating system.[16] Apart from spreading, the *Morris worm* did nothing else. However, since a proper stopping mechanism was not included, the worm spread rapidly—infecting the same hosts

[15] https://www.f-secure.com/v-descs/cih.shtml.
[16] https://limn.it/articles/the-morris-worm/.

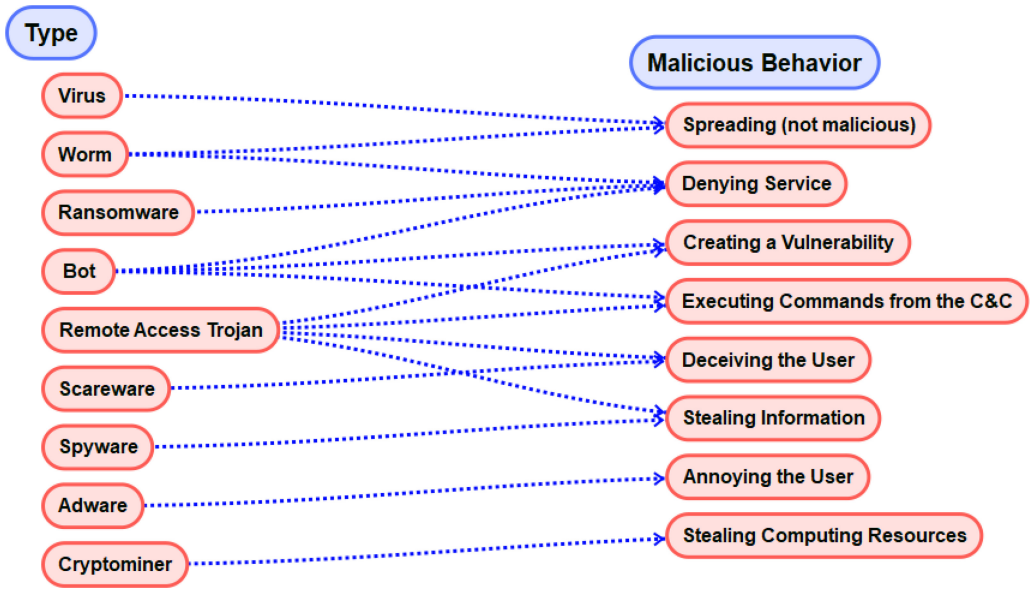

Fig. 2. Malicious behavior associated with each malware type.

multiple times. This created a *denial-of-service* attack, which caused a slowdown in performance of the network and the victim hosts.

However, some non-malicious software can spread, such as the *Linux.Wifatch*, a worm-like program that infects vulnerable IoT devices with weak or default passwords through Telnet.[17] After infection, the program removes any malware it finds from the device, closes the Telnet port, and prevents further infections. This example demonstrates that spreading is not necessarily malicious. To avoid confusion, we will not refer to this behavior as malicious in this survey.

The behaviors listed above are not mutually exclusive. For example, a malware might steal computing resources as well as execute commands from the C&C (e.g., to update its code). It might also steal passwords and open a backdoor for later infection. Classifying malware by behavior (instead of by type) is beneficial in two ways: knowledge of the behaviors found in a malware helps us improve our analysis tools, and it is easy to correctly classify a malware by type once we know how it behaves. Figure 2 shows the malicious behavior that can be seen in each malware type. This diagram demonstrates the correlation between the classic malware taxonomy (Section 3.1) and the behavioral taxonomy we presented here. For clarity of the diagram, the order of items is different than the order they were presented.

3.3 Classification of Malware by Privilege

In addition to the behavioral classification of malware, attention should be drawn to the privilege malware runs at during execution. As malware gains more privileges it can do more damage, and it is harder to detect and analyze. Analysis of unknown binary code poses a risk of infection. In the worst-case scenario, the analyzed code infects the system beyond repair, without the user's knowledge. Such infection could result in a false report, misclassification of a sample, or even compromise the entire organization.

[17] https://www.symantec.com/connect/blogs/there-internet-things-vigilante-out-there.

The Intel x86 CPU is built based on four protection rings, numbered from zero to three with Ring 3 being the highest, and Ring 0 being the lowest [25]. Rings 1 and 2 are not used by Windows, and thus will not be covered in this survey. Code running at a lower protection rings has more privileges (e.g., read/write permissions) over code running at higher ones. The four levels of privileges are based on these protection rings:

User mode (Ring 3)—When a new process is started, its code is loaded into the RAM with *user mode privileges*. Any code that does not require more than user mode privileges is considered *user mode code*. In Windows operating systems, this includes any software installed by the user, as well as large parts of the operating system itself (even the *Administrator* account has only *user mode privileges*). When analyzing *user mode* malware, infection can be cleaned easily by undoing the changes made by the malware or by reformatting the system completely.

Kernel mode (Ring 0)—The kernel is the part of the operating system responsible for handling the system resources. It provides functionality for communicating with the hardware, and it is responsible for managing all aspects of the system (memory, networking, process priorities, CPU time, etc.). The kernel runs at Ring 0 with *kernel mode privileges* (aka *root privileges*). This protection ring is reserved only for the OS kernel and system drivers. It allows the operating system to control the physical devices, manage resources such as CPU time or memory allocation, and control *user mode* code. *Kernel mode* code can load new code into the kernel when necessary, for example, when new hardware is connected to the system (like a camera or USB storage device). This type of code is called a *driver*, and it provides the necessary functions to allow interaction with the new device. Malware might gains access to the system's kernel to perform operations with *root privileges*, and thus this type of malware is also called a *rootkit* [26–28].

Hypervisor (Ring -1)—Hypervisor technology enables the execution of several virtual operating systems simultaneously on the same physical hardware. A hypervisor runs with more privileges than *kernel mode*; thus, is said to be running in Ring -1, even though this is not an actual protection ring. There are two types of hypervisors: *Type 1 hypervisors* support multiple operating systems running in parallel (see Section 6.3). Malware authors have long realized the potential of hypervisors. For example, a malicious hypervisor can be installed to trap the operating system in a virtual machine (VM) and take away its root privileges [29]; in this way, the malicious hypervisor gains superiority over the kernel, effectively giving it control of the operating system. Any analysis tool installed on the OS will be unaware of code executed by the hypervisor. Malware that installs a malicious *type 1 hypervisor* is called a virtual machine-based rootkit (*VMBR*) [30, 31]. *Type 2 hypervisors* allow the execution of a virtual machine (see Section 6.2).

Hardware (Ring -3)—Infecting a hardware device means that the malware can run freely without fear of detection, and launch attacks against other devices from outside the CPU (such a malware is sometimes referred to as Ring -3 rootkit [32]). Malicious firmware update [33] is a common attack to achieve *hardware privileges*. Every hardware component includes code (*firmware*) that operates the device. The firmware can be updated from time to time to fix bugs and patch security vulnerabilities. However, if a vulnerability in the update process is discovered by an attacker, then the vulnerability could be used to install malicious firmware that is hidden from the CPU and can be used to launch an attack on the system (see Section 4.1). Hardware infection is common in USB [34], IoT, and medical devices.

3.4 About Behavior and Privilege

The two taxonomies we presented here (behavior and privilege) are different from the traditional taxonomy in one key way: Malware can usually be classified into a single type (meaning that most of the time malware types are mutually exclusive); however, this is not the case for classification by behavior or privilege, which are not mutually exclusive. A malware sample can display more than

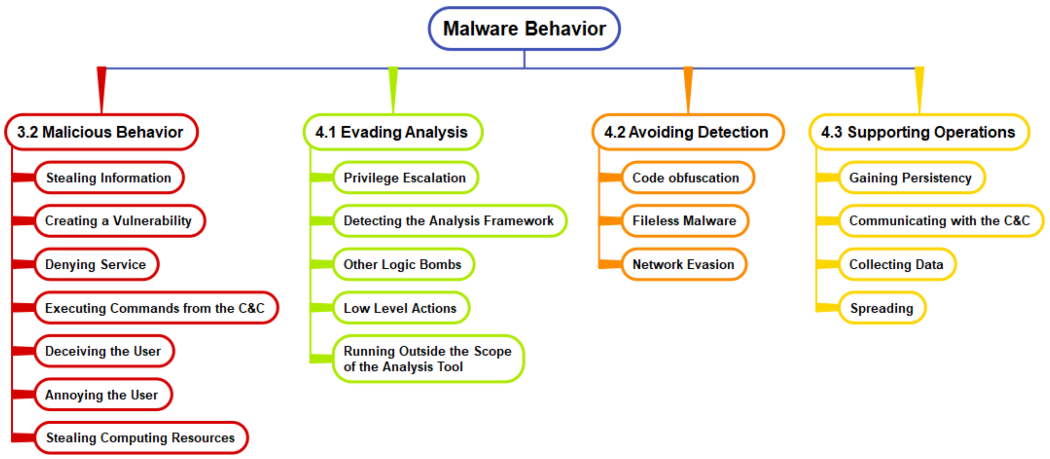

Fig. 3. Malware Behavior Taxonomy.

one behavior. Identifying these behaviors is the first step toward improving the analysis process and protecting systems from infection. In addition, any combination of behavior and privilege is possible: stealing information by infecting hardware devices, using rootkit to create a denial-of-service attack, stealing computing resources with user mode privileges, and so on.

4 ANALYZING MALWARE BEHAVIOR

To understand the problems, one might encounter during malware analysis, one must have a good idea regarding the behavior malware might exhibit. Section 3.2 listed the behaviors malware employs to perform malicious tasks, but there are other behaviors malware uses to evade analysis or hide its presence. We present these behaviors in Figure 3. Note that the numbering of titles appear in the figure corresponds to the relevant section for the convenience of the reader.

4.1 Evading Analysis

Privilege escalation—Malware can evade analysis by moving code from one protection ring to a lower one (by loading a malicious kernel driver, tricking the user into providing the malware with elevated privileges,[18] installing a malicious hypervisor, or infecting a hardware device). Having more privileges allows malware to perform its malicious tasks with less risk of detection or analysis (see Section 9). Malware can gain any of the privileges we presented in Section 3.3:

Kernel—A common technique used by *rootkits* to infect the kernel is by loading a malicious kernel driver. This code provides the malware with root privileges, allowing it to hide its presence, and subvert in-guest analysis components.

Hypervisor—Since hypervisors run in a more privileged mode than the kernel, it is possible for the hypervisor to trap all Ring 0 operations. If a malicious hypervisor is installed, then it gains control of the kernel "from the outside." Thus, without manipulating the OS, the malicious code is able to run undetected. Any analysis tool installed on the OS will be unaware of code executed by the hypervisor. This attack is the *VMBR* mentioned in Section 3.3.

Hardware—Infecting hardware devices is very effective, although it is the most complicated technique of all. A great deal of skill and knowledge are required to find and exploit a hardware vulnerability, but once achieved it allows the malicious code to run outside of the CPU, thereby

[18] https://nakedsecurity.sophos.com/zeroaccess/.

evading any analysis tool that might be present in the system. Attacking hardware devices can be done using a malicious firmware update (such as flashing the BIOS).

To allow faster processing of I/O operations, modern hardware architecture provides *direct memory access* (DMA) to the system's RAM. This enables some hardware components (such as the network interface card) to perform their operation without requesting CPU time, thus reducing the load on the CPU. While this feature utilizes the system resources to enhance performance, it can be exploited by hardware privilege malware using a technique called a *DMA attack* [35].

Detecting the analysis framework—One of the most effective ways for malware to evade analysis is by hiding when it detects that it is under analysis. Since only executed code can be analyzed, a malware can subvert its normal execution path and hide its malicious activities (e.g., instead of installing a keylogger, the malware can simply terminate itself). In this case, the analysis report produced will not include any trace of the malicious behavior, and the file could be falsely flagged as benign. Here are a few examples of how malware detect the analysis framework:

- ***Traces of the analysis tool***—Analysis tools leave many fingerprints on the system (such as the names of processes, installed drivers, registry keys, and more [36, 37]), which are used by malware to detect their presence. In a way, malware use signatures to detect analysis tools just like security applications use signatures to detect malware. Detection of specific analysis components can come in many forms and gets highly creative.[19] VM and hypervisors are commonly used for malware analysis, making their presence a strong indication of an analysis framework. Over the years many techniques have evolved to detect the virtual machines such as searching the location of the Interrupt Descriptor Table (IDT), flushing the translation lookaside buffers (TLBs), searching for hard-coded MAC addresses, executing VMX code, and so on [38, 39].
- ***Debuggers***—The presence of a debugger (Section 7.2) is another indication for malware that it is under analysis. Being a common method for software analysis, the presence of a debugger is a definite indication that malware analysis is taking place.
- ***Time-based detection***—Since at least some overhead is required when analyzing code dynamically, the most obvious way to detect analysis is to measure the time it takes to execute specific segments of code. By comparing the duration of some specific operation to its "normal" execution time, the malware can expose the existence of an analysis framework.
- ***Traces of users' activities***—When malware infects a personal computer, the malware can tell whether a user has been using their computer for some time, since this leaves traces on the system. Analysis environments, however, are frequently wiped clean to allow "a fresh start" between analyses of different malware. The lack of traces of users' activities is sometimes used by malware to detect an analysis environment.[20] Checking if the browser's history (or the recent documents list) is empty, detecting a "fresh" operating system, testing if the window of the process is in focus, checking the number of screen monitors connected, checking the browser history, testing for mouse movement, and so on, have all been seen in malware.[21]

Other logic bombs—A *logic bomb* is a piece of code designed to test various conditions about the environment. If those conditions are met, then it triggers a malicious behavior. Many of the previous examples are basically logic bombs; other examples of logic bombs are searching for

[19] https://www.rsaconference.com/writable/presentations/file_upload/hta-w10-understanding-and-fighting-evasive-malware_copy1.pdf.
[20] https://www.lastline.com/labsblog/malware-evasion-techniques/.
[21] https://litigationconferences.com/wp-content/uploads/2017/05/Introduction-to-Evasive-Techniques-v1.0.pdf.

specific software/hardware, waiting for a specific event (e.g., system shutdown signal), waiting for a command from the C&C Server, or waiting for a specific date or time.

Low-level actions—To evade analysis, malware authors utilize *undocumented kernel functions* exported by the operating system to perform their malicious actions. These low-level functions are implemented by the Windows kernel and can be invoked by *user mode* processes. *Kernel functions* are designed to be used only by the operating system itself, and thus these functions are not documented. Another method of evading analysis is by directly manipulating components of the OS (e.g., the registry) without calling the proper Windows API functions. This technique requires advanced skills and a deep understanding of how the OS tracks data, where the data is stored, and so on. However, a successful attack implementation is very hard to detect and analyze.

Running outside the scope of the analysis tool—There are various ways this can be accomplished, including:

- ***Code injection***[22]—To avoid analysis and detection, malware can also hide by running malicious code from a different process (aka *code injection*). By injecting code into other processes, the malware can run its code as if it was an integral part of a legitimate process. Most detection tools contain a list of trusted processes (whitelist). These processes are usually unmonitored and might have elevated privileges. Thus, malware authors try to hide their malware by finding clever ways to run their code from whitelisted processes. Here are a few examples of *code injection* techniques: (1) DLL injection—Using the Windows API, malware can allocate memory on behalf of another process and inject it with a precompiled DLL file. This DLL file can start its execution as a new thread inside the remote process, and every instruction performed by the DLL will be part of the attacked process. This technique is also used to gain higher privileges. (2) DLL search order hijacking—When a process tries to load a DLL file by its name, the operating system searches for that file in specific locations (such as the process folder, c:\windows\system32\, and others). Placing a malicious DLL with the same name in one of these locations can force the OS to load a malicious code instead of the actual DLL. (3) Process hollowing—A technique that takes over an entire process by unmapping the original code from the memory and injecting malicious code into the "hollowed out" process. The structure of the original process is left intact, meaning that the identity and all permissions of the process are owned by the malicious code. Other techniques for *code injection* include *atom bombing, thread execution hijacking*, and *COM hijacking*.
- ***Infecting the host***—If a vulnerability in the system is found by attackers, which allows malware to escape the guest OS and execute malicious code on the host, then the analysis can no longer analyze the malicious code. This is the worst-case scenario for the analysis, since it is very hard to detect and clean host infections.
- ***Nested virtualization***—Malware sometimes create a virtual machine inside an infected system and execute the malicious code inside the VM, behind the semantic gap (see Section 7.4). This allows the malware to evade detection and analysis.

4.2 Avoiding Detection

Code obfuscation—Traditional detection methods commonly used by anti-virus software rely on scanning the hard drive for known malware. Anti-virus software detects known patterns (signatures) of malicious code and quarantine or delete suspected files. To hide from anti-virus scanning

[22] https://www.endgame.com/blog/technical-blog/ten-process-injection-techniques-technical-survey-common-and-trending-process.

and other detection tools of this sort, malware authors utilize various techniques to hide strings. *Code obfuscation* [40] is commonly used by malware authors to prevent extracting signatures derived from the malware's binary code. Here are a few examples for *code obfuscation* techniques:

- ***Polymorphism (aka Packing)***—By encrypting the entire code, malicious authors can hide not only the strings but also the binary code of the malware. After writing and compiling their malware, attackers sometimes envelope their binary code with an encryption mechanism (a process called *packing*). Encryption randomizes all of the binary data, thwarting any future attempt to create a signature from the malware's binary code. Each sample of the malware is packed with a different encryption key, which is also included in the sample. A decryption method is wrapped around the encrypted code and executed when the process is loaded. At run-time, the decryption method uses the key to decrypt the data and load the malicious code into the memory or on the hard drive (this process is called *unpacking*). When the unpacking process has been completed, execution is handed over to the malware to perform its operations. A method for detecting polymorphic malware was presented in Reference [41].
- ***Metamorphism***—The second type of code obfuscation is *metamorphism*. To create a new copy of the malware the *metamorphism* process rearranges the malware's binary code; inserts garbage opcode instructions; changes the registers used; and so on. This makes each copy of the malware different from previous copies, however all of these copies perform the same operations. This process makes it harder to extract a signature for future detection, although a method for detecting metamorphic malware was presented in Reference [42].
- ***Deep neural networks***—Recently, researchers at IBM presented a new obfuscation technique using neural networks.[23] The researchers have used hidden layers of a deep neural networks to hide the conditions for the activation of the malware. Due to the complexity of deep neural networks, it is currently not possible to reverse engineer such networks, thus making it very useful for code obfuscation.
- ***Domain generation algorithm*** (DGA[24])—Many signatures are based on the strings the malware contains. These strings can be easily obtained using designated tools such as *strings2*[25] or any debugging tool. Any unique string (like URL addresses) found in the malware's code can be used as a signature. The ease with which this can be done led malware authors to hide signatures such as URL addresses used by the malware. This can be accomplished with a *domain generation algorithm* (DGA), an algorithm that generates a domain name upon request, which can be purchased by the attacker. Using DGA allows the malware to communicate with different C&C addresses, without including these addresses in the malware's strings, and thus cannot be used as signatures by detection tools. Another advantage this technique offers to the malware is that the generated addresses cannot be blocked in advance by victims.

More information about malware evasion techniques can be found in Reference [43].

Fileless malware—Another approach that can be taken to avoid detection is to run the malware without writing anything on the hard drive [44]. Most traditional detection methods (mainly antivirus software) rely on scanning the hard drive and looking for known malware or signatures. Avoiding a presence on the hard drive enables the malware to avoid such detection methods and increase its stealth abilities. A fileless malware only lives in the memory, and there is no evidence

[23] https://securityintelligence.com/deeplocker-how-ai-can-power-a-stealthy-new-breed-of-malware/.
[24] https://blog.malwarebytes.com/security-world/2016/12/explained-domain-generating-algorithm/.
[25] https://github.com/glmcdona/strings2.

of its presence on the hard drive. Any executable loaded to the RAM will stay there until it is terminated or the system shuts down. This is an advantage when infecting servers (which are rarely shut down) but is an issue when infecting personal computers. To load itself again after a system shutdown, the malicious executable must *gain persistency* (see Section 4.3). The fileless approach is common among cryptominers,[26] since they are often used to either infect servers, which as mentioned above are seldom shut down, or infect users via the Web browser and remain fileless,[27] thus avoiding detection by most anti-viruses.

Network evasion—Networking activities can also be used as an indication that a malware has infected the operating system. Denying or restricting network access is often used to thwart malware (using firewalls to block outside connections to sensitive networks is a commonly means of accomplishing this). To bypass such protection mechanisms, the malware might employ any combination of the following network evasion techniques [45]:

- ***Reverse connection***—Firewall protection can be bypassed by having the attacked endpoint open a connection back to the C&C server or the attacker. This is possible because most of the time firewalls are not configured to block outgoing connections. Since blocking all outgoing connections completely is not acceptable in most organizations, this technique proves itself very useful for attackers.
- ***Encrypted sessions***—can be used to hide the nature of data sent to the C&C. Common encryption protocols (such as SSL) utilize strong encryption algorithms and cyphers that cannot be broken without advanced resources and a significant amount of time, allowing the malware to hide for a longer period of time.
- ***Tunneling***—Sending data covertly is possible using unconventional methods such as DNS or ICMP requests. For example, DNS tunneling sends information over innocent looking DNS requests.[28] By sending multiple DNS queries, the malware is able to send data to its C&C without establishing an actual connection that might raise suspicion. Since many monitoring tools ignore DNS requests, data can be sent back to the attacker without being detected.

4.3 Supporting Operations

The functionality needed for the malware to execute its malicious payload is presented here. Most of the behaviors described in this section can also be found in benign applications such as anti-virus software and other security mechanisms.

Gaining persistency—Malware needs to find a way to load itself after each reboot. The Windows registry can be used for this purpose, but other techniques are also used, including: *shortcut modifications, DLL search order hijacking*, subverting the boot process and loading a malicious kernel (such malware is called a *bootkit* [46, 47]), hardware infection, and others.[29] One method malware use to *gain persistency* is by adding itself to an *Auto-Start Extendibility Point* (ASEP),[30] such as the registry startup key in Windows. For example, adding a malicious command to the *Run* or *RunOnce* registry keys will cause the computer to execute that command when the system has finished the booting process. Using malicious scripts, *fileless malware* (Section 4.2) can force their malicious code to be loaded back into RAM every time the operating system starts, without having a file on the hard drive.

[26] https://blog.minerva-labs.com/ghostminer-cryptomining-malware-goes-fileless.
[27] https://www.forcepoint.com/blog/security-labs/browser-mining-coinhive-and-webassembly.
[28] https://resources.infosecinstitute.com/dns-tunnelling/.
[29] https://www.andreafortuna.org/dfir/malware-persistence-techniques/.
[30] https://www.microsoftpressstore.com/articles/article.aspx?p=2762082.

Communicating with the C&C—Communication with the C&C includes getting commands from a handler, receiving updates, and sending extracted data. Malware communication is usually covert, enabling malware to bypass detection tools and firewalls (see Section 4.2).

Collecting data—Malware collect data from various sources, including: volatile memory (e.g., passwords, credentials, and cryptographic keys), the file system (e.g., documents), the operating system (e.g., registry values, active processes, system configuration, open windows), user activities (e.g., websites visited, open documents, keystrokes), and the network (e.g., connected printers, open ports on other hosts).

Spreading—Spreading methods include sending infecting emails (via attachments or malicious scripts); exploiting vulnerabilities to allow remote execution; file sharing (e.g., P2P file sharing, torrent clients, shared folders); and the use of Bluetooth, USB, and other portable devices. By itself, spreading is not malicious (see Section 3.2).

4.4 Dynamic Analysis Properties

To conclude this section about malware analysis, we summarize the required characteristics of any malware analysis process to minimize the risk of infection and produce accurate results:

- **It must be trusted**—Data provided by the analysis framework must not be compromised by the malware [48, 49]. A defensive mechanism must be in place to prevent the malware from gaining control of the system.
- **It must be undetectable by the analyzed file**—If malware can tell that it is being analyzed, then it might terminate itself or execute only non-malicious commands [6].
- **It must collect as much relevant data as possible about actions performed by the malware**—Such information can include functions calls, parameters, networking, file system modifications, hardware measurements, and so on [1].
- **It must meet the malware's expectations to expose its full behavior** - The relevant operating system(s) must be installed, the appropriate hardware must be connected, and the vulnerable application must be installed.
- **It must limit/emulate network access by the malware**—Allowing malware to connect to internal networks or the Internet must be limited and only take place under supervision to prevent infection of other devices or exfiltration of sensitive information [80].
- **It must generate a coherent and concise report**—The information in such report can be utilized to produce decision regarding the classification of the analyzed file, either by security expert or machine-learning algorithm.

Despite all of its benefits, analyzing malware dynamically has a few limitations:

- Only executed code is observable. This means that if a required condition is not met precisely, then some code might not be executed and thus go unanalyzed.
- Dynamic analysis also requires computational overhead, which may slow down execution.
- The analysis must be performed on the specific OS and/or hardware targeted by the malware.

5 DESCRIPTION OF DYNAMIC MALWARE ANALYSIS FRAMEWORKS

In this section, we will present the various components of frameworks designed to dynamically analyze malware. Various malware behaviors are designed to detect or evade analysis (Section 4), and these behaviors must be taken into account when developing an analysis framework. It is also imperative to know as much as possible about the privilege level the malware runs at and protect from its malicious behavior (Section 3.3). For example, when analyzing code that performs

a firmware update to the BIOS with malicious code, the analysis framework must protect itself from this. A failure to do so means that the malware will be able to manipulate the BIOS directly, compromising the integrity of the entire system.

Trusted and secured analysis of malware means that the analysis tool and the malware are not competing for the same resources [48, 49]. A framework can be considered trusted if it meets the following two conditions:

Protecting against infection—Any action performed by the malware should be executed and assessed to gain full understanding about the malware and its affects, while protecting the analysis system. Accomplishing these contradicting tasks is possible by allowing the malware to infect only a portion of the analysis framework. One way to protect against infection is to use guest-host model, meaning that the malware and analysis tool are executed on separate operating systems. This is possible by using a *virtual machine* (Section 6.2), a *hypervisor* (Section 6.3) or an *emulator* (Section 6.4). The guest-host model creates a separate execution space (RAM and CPU time) for each of the operating systems. When using such guest-host model, the analysis system must ensure that only the guest OS is infected. If the malware is capable of escaping the analysis environment and infecting the host OS (Section 4.1), then the analysis framework can no longer be considered trusted. *Volatile memory forensics* (Section 6.5) or *side-channel analysis* (Section 6.6) are also used to prevent the malware from infecting the entire system.

Avoiding detection—To prevent malware from detecting that it is under analysis (Section 4.1), the framework and all of its components must be hidden. If, for example, the analysis tool is not adequately hidden, the malware might detect the framework's presence and subvert its behavior. Preventing malware from detecting the analysis framework requires that no footprints are left by the framework (such as analysis processes, drivers, hard-coded hardware components, registry keys, special opcode instruction sequences, etc.) Countless methods have been used by malware to detect analysis frameworks, creating an arms race between malware authors and analysis tool developers. A GitHub repository called Pafish implements various detection methods used by malware for testing analysis platforms.[31]

All software analysis must start on a clean system, without the presence of any malicious code. However, because dynamic analysis is based on executing potentially harmful code, great care should be given to the robustness of the analysis system. After analysis is performed, it is likely that the analysis system contains some malicious code and/or has been compromised; therefore, the system must be returned to a clean state so a new sample can be analyzed. Returning a bare metal system to a clean state can be accomplished by completely replacing the hardware or reformatting the system. However, since this solution cannot be automated, it is not practical for large-scale analysis. The host-guest model is a more effective approach for returning the system to a clean state. In this case, the state of the guest OS can be saved to a file (called a *snapshot*) for later use. Snapshots can be saved to the disk on the host, or on some other storage device. Snapshot files includes the RAM and file system of the guest, thus comprehensively representing a machine's state. By returning the machine to a clean state before performing analysis and then again in between samples, ensures that any malware traces will be completely erased.

A dynamic malware analysis framework is composed of three components:

- ***Malware sample***—This can be an executable code, a script, a document, or a firmware.
- ***Hardware and operating system***—Malware expect to be executed on a specific OS or hardware component. If the analysis framework does not meet these expectations, then the malware will not execute.

[31] https://github.com/a0rtega/pafish.

Fig. 4. Bare metal analysis layout.

- ***Analysis tool***—A software or device used to monitor the analyzed code and/or system. The analysis tool should produce a report summarizing the malware's behavior. The usefulness of a report is based on the amount of knowledge it provides. In general, higher abstraction is preferred for classifying the sample (i.e., determining whether it is a malware or not, and which type of malware it is). However, to fully understand the malware behavior and techniques, lower abstraction is needed (i.e., determining which parts of the system were affected by the malware and how, which system calls that were executed, etc.). Including raw data such as a network PCAP file is also useful for in depth analysis (when needed).

A key term we will use in this survey is the **analysis layout**. It refers to the way these three components are arranged in relation to one another. Several *analysis layouts* are possible, including *bare metal, virtual machine, hypervisor, volatile memory acquisition*, or *side-channel data acquisition*. Each of these analysis layouts is presented in Section 5.

6 ANALYSIS LAYOUT TAXONOMY

6.1 Bare Metal

Unpacking a new PC, installing an operating system and analysis tool, and infecting it with malware is an analysis environment that can be called *Bare Metal* (Figure 4). The analysis process might run in *user mode* (Ring 3) or *kernel mode* (Ring 0). Considering that the malware might be a rootkit that could disrupt the analysis application or infect the system beyond repair, this solution is only useful when analyzing *user mode* malware.

Returning the system to a clean state can be done by undoing the malware effects (for *user mode* malware) or by reformatting the system and installing a fresh, reliable, operating system (for *rootkits*). Cleaning the native OS is done manually, and thus it is very time-consuming. A solution to this problem, in which trusted snapshots are created on bare metal, was presented by Reference [50].

6.2 Virtual Machine

Running the analysis tool and malware inside a virtual machine (also called a *type 2 hypervisor*) can protect the native system (the *host*). This layout is based on the separation between the host OS and the guest's kernel, which protects against hostile takeovers. Resetting the guest VM to a clean state can be accomplished by taking a snapshot before executing any malware and reverting to that snapshot after each analysis. One possibility for analyzing malware using a virtual machine

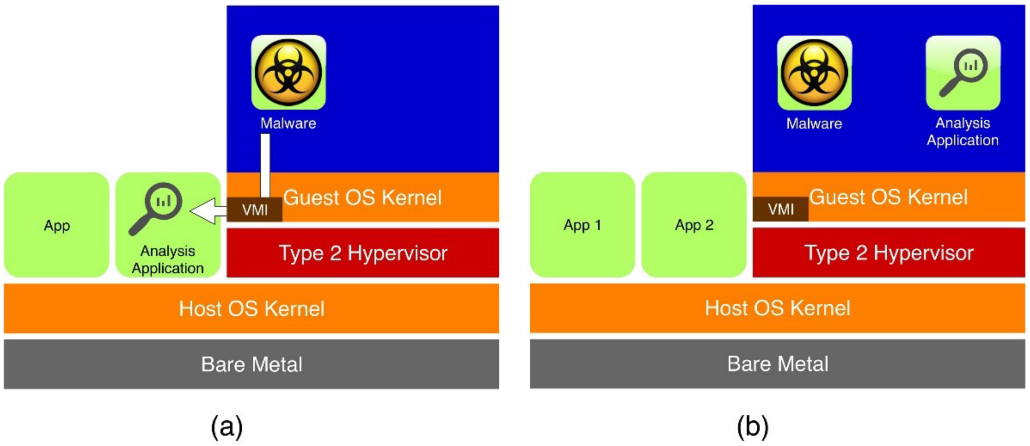

Fig. 5. Virtual machine layout: (a) analysis tool inside the guest; (b) analysis tool on the host.

is to include the analysis tool inside the virtual machine along with the malware [Figure 5(a)]. This layout is still limited to analyzing *user mode* malware, but it improves upon the bare metal layout by providing a fast and reliable resetting mechanism.

Another possibility is to install the analysis tool on the host and track the changes made by the malware "from the outside" [Figure 5(b)]. Learning what is happening inside the virtual machine is done using a technique called virtual machine introspection (VMI), which uses an in-guest component (usually a kernel driver). By querying the VMI component from the host, the analysis tool can gather information about how the malware affects the operating system. However, if the VMI component can be detected by malware, then the presence of the virtual machine is exposed, and the analysis can no longer be trusted.

On rare occasions, a vulnerability in the virtual machine implementation is exposed, allowing the host system to be infected from inside the VM. In such cases, the entire analysis platform is compromised and cannot be trusted until the infection has been completely removed. When such a vulnerability in the VM implementation is found, it is crucial to patch the system as soon as possible to maintain the integrity of the analysis framework.

6.3 Hypervisor

A third layout option is using a *type 1 hypervisor* (Figure 6). Hypervisor technology enables the execution of several operating systems in parallel on the same physical hardware (aka *virtualization*). Each operating system is loaded onto a VM, with a separation between them. To enforce such a separation, two kernel modes are introduced in this layout option: *VMX root* (sometimes owned by the hypervisor itself) and *VMX non-root* (for any guest OS kernel).[32]

This layout limits the guest OS's access to hardware resources (e.g., when the guest kernel tries to read or write from an external device). A guest kernel trying to access some hardware device triggers a *VMEXIT* operation, which transfers control to the hypervisor (*VMX root*). After gaining complete control of the system in this manner, the hypervisor can deny or allow this request. Once the operation is complete, the hypervisor can then return the execution back to the kernel using a *VMENTRY* operation. Resetting the system to a clean state is also handled using snapshots.

[32] https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf.

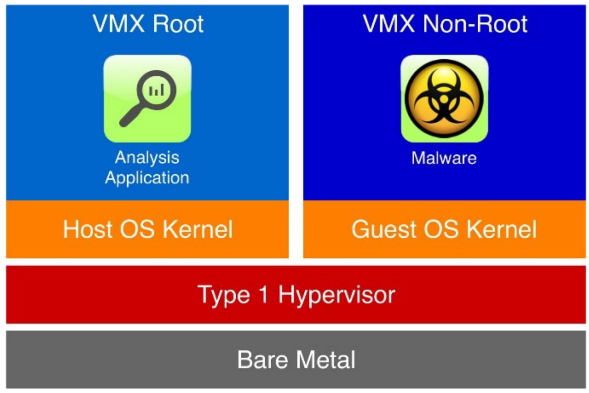

Fig. 6. Hypervisor layout.

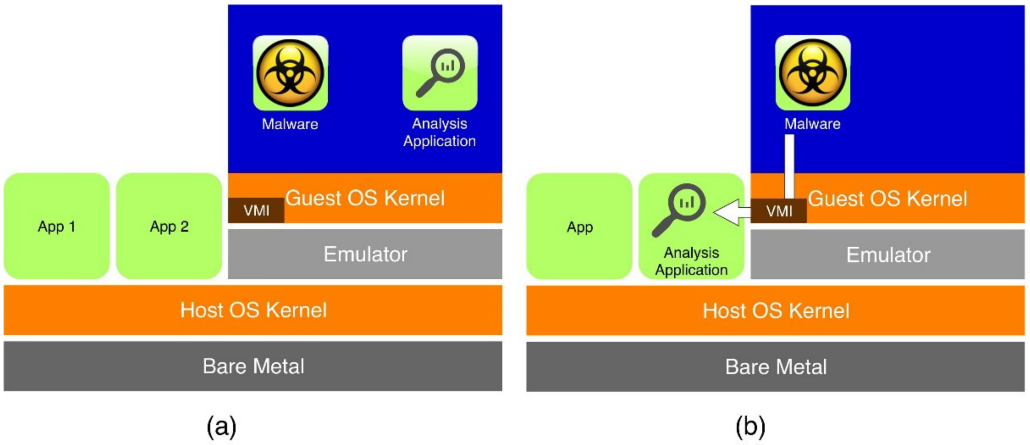

Fig. 7. Emulator layout (a) analysis tool inside the guest; (b) analysis tool on the host.

6.4 Emulation

Instead of relying on protection mechanisms to prevent host infection, it is possible to separate the malware completely from the hardware using *full system emulation* to gain knowledge about the malware behavior (Figure 7). An emulator is a piece of software that provides a hardware level implementation of a computerized system (such as the states of CPU registers, physical memory, and I/O devices) and does not execute malware on the physical hardware. Analysis frameworks using this layout might include an in-guest component to track the changes made by the malware [Figure 7(b)], which makes them a target for detection and subversion by malware. When emulation takes place without an in-guest component, *bridging the semantic gap* is necessary (see Section 7.4). In addition to analyzing personal computers, it was shown that emulation can also be used for firmware analysis [51].

6.5 Volatile Memory Acquisition

Volatile memory forensics is a methodology for analyzing of the OS's volatile memory (RAM) without the need for an in-guest component. *Volatile memory forensics* is composed of two steps: ***acquisition*** and ***analysis***.

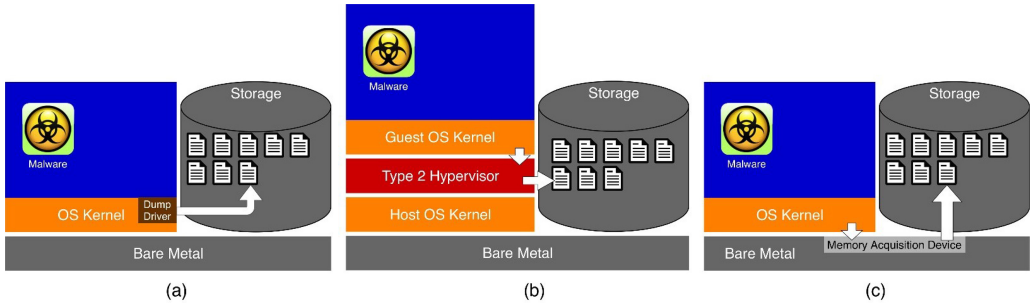

Fig. 8. Volatile memory acquisition: (a) using kernel driver on bare metal; (b) using a VM (type 2 hypervisor); (c) using hardware.

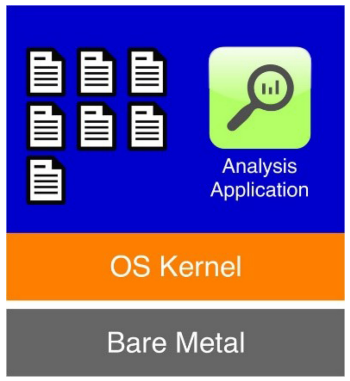

Fig. 9. Volatile memory analysis on a separate machine.

Volatile memory acquisition is the process of copying the RAM to a memory dump file, which can be stored on the host or external storage device. Acquisition can be performed using any of the previous layouts presented in this section, by either software or hardware:

Volatile memory acquisition by software—On *bare metal* layout, the acquisition can be performed by a kernel driver [Figure 8(a)]. It is also possible to acquire memory dumps from a VM by a *hypervisor* [Figure 8(b)] or *emulator*.

Volatile memory acquisition by hardware—A special hardware component installed on *bare metal* is another option for volatile memory acquisition (Figure 8). It permits suspending the operating system and accessing the physical memory directly.[33] Since this process is done at the hardware level, outside the scope of the operating system and everything within it, the malware should not be able to detect or interfere with the dumping process. However, that is not always the case [52].

When properly performed, *volatile memory acquisition* does not leave any footprints in the system [53, 54]. Assuming the dumping is performed without an in-guest component (either by hardware or by dumping the RAM of a guest OS), the malware has no way of telling that a dump was taken mid execution. Of course, this is only true in those cases where the dumping process can be performed quickly enough to prevent any *time-based detection*. Another benefit of *volatile memory forensics* is that the analysis process itself can be done on a separate machine (Figure 9).

[33] CaptureGUARD - http://www.windowsscope.com/product/captureguard-gateway-access-to-locked-computers/.

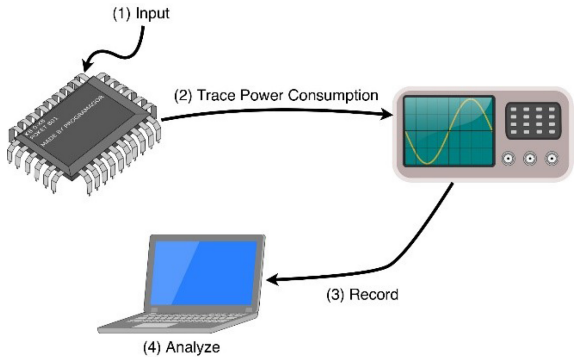

Fig. 10. Side-channel data acquisition and analysis using an external measuring device.

The second step in *volatile memory forensics* is ***volatile memory analysis***. This is the analysis technique designed to extract useful information from the memory dump file and detect the presence of malware and its behavior (see Section 7.4).

The drawback of *volatile memory forensics* is that the analysis is not continuous. A memory dump can be taken only once in a while, which prevents the analysis of single operations (see Section 7.2). However, memory dumps permit examination of the way in which the system changes between one dump to the next. Therefore, this methodology is suitable for analyzing malware that reside in the memory for long term (like RATs and cryptominers). If the malware starts and terminates before the next dump is taken, then the memory space assigned to the malware might be assigned to a different process. In such case there might not be any traces of the malware in any of the dumps.

6.6 Side-Channel Data Acquisition

Side-channel analysis is a methodology to analyze an electronic device by monitoring the physical power consumption and electromagnetic (EM) emission. Originally, it was used for crypto analysis (the process of extracting private keys used during encryption in microcircuit devices) and referred to as a *side-channel attack*.[34] Similar to *volatile memory forensics*, the acquisition and analysis of data can be done separately.

Side-channel data acquisition using hardware—Connecting a measuring device to the *bare metal* to track the power consumption or EM emissions of the device under analysis (Figure 10). This is extremely useful for analyzing IoT devices.

Side-channel data acquisition using software—Modern CPUs include special registers called *hardware performance counters* (HPCs). The HPCs tracks internal CPU events, and by querying them often a behavioral pattern of the system's operation can be produced. The HPCs can be measured using an in-guest component (e.g., kernel driver) or via the *hypervisor*.

Side-channel data analysis—Analysis techniques of the side-channel information acquired is presented in Section 7.5.

In terms of detection by malware, *side-channel data acquisition* is more trusted than any of the other layouts presented here. Data acquisition without using an in-guest component is entirely transparent to the malware as well as the OS. Also note that if the analysis is performed on a separate machine, then it creates almost no overhead compared with normal execution, thus reducing the risk of *time-based detection*. If, however, the HPCs are measured using an in-guest component,

[34] http://gauss.ececs.uc.edu/Courses/c653/lectures/SideC/intro.pdf.

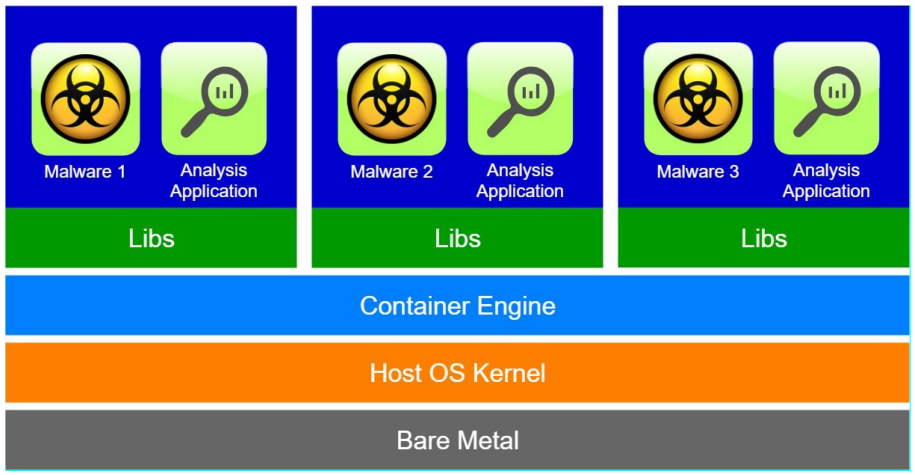

Fig. 11. Containerized environment layout.

some overhead may result, allowing *time-based detection* by malware. The in-guest component itself is also vulnerable to manipulation by rootkits and thus it cannot be considered trusted.

6.7 Containers

Recent years have seen a rise in the use of containerized environments. This technology trend is becoming more and more widespread as it offers a new separating layer between the application and the OS.[35] Such a separation allows software developers to be indifferent to the target OS and develop a single version of their code, which can be executed on all operating systems. In addition, *containers* offer better usage of cloud computing by running many "micro-service" apps instead of a large, monolithic process.[36] *Containers*, unlike virtual machines, share the OS kernel with the host. In addition, *containers* are not completely separated from one another (Figure 11), this puts into question the security of containers for malware analysis.

We believe that *containers* are here to stay and might play a significant role in future malware development and/or malware analysis. To the best of our knowledge though, it has not yet been determined whether or not *containers* can be well used by malware to infect victim hosts. While there have been a few suggestions for using *containers* for malware analysis (such as REMnux[37] and [55]), it may be wise to wait until its use in this area advances before including it in a survey.

7 ANALYSIS TECHNIQUES AND ACADEMIC TOOLS

In this section, we review different analysis techniques (marked with ***italic, bold, and underscored text***), as well as analysis tools developed as part of academic research (marked with ***italic and bold text***), some of which have been developed using commercial or open-source platforms. These platforms (marked with *italic and underscored text*) are presented in the Appendix.

7.1 Function Call Analysis

Every process relies on function calls to perform its duties, whether these functions are internal to the process, or external (e.g., functions exported by other processes, *system calls*). The behavior

[35] https://www.infoworld.com/article/3204171/what-is-docker-docker-containers-explained.html.
[36] https://developer.ibm.com/articles/why-should-we-use-microservices-and-containers/.
[37] https://remnux.org/.

of the analyzed malware can be better understood by tracking the various functions called by the malware and the parameters correlating with these functions. Obtaining notification when a function is called can be achieved by ***hooking*** a piece of code to that function. Common hooking techniques include the Windows OS hooking mechanism (which notifies the analysis process when a function is called[38]), and various *code injection* techniques (Section 4.1). The injected code is mostly added to the beginning of the function and when executed it notifies the analysis process that the hooked function was called. Once a hooked function is called, the analysis process can then access the process's stack and obtain information about the function called and the parameters passed to it. Other useful information can be gathered from the OS for increased understanding of the context in which the function was called.

TTAnalyze [56] is an analysis tool extending the *QEMU* emulator with an in-guest component referred to as *Inside the Matrix* (*InsideTM*), which translates virtual addresses to physical ones. This step is important, because identifying the process to which a memory page belongs can be a time-consuming and challenging task, which can be simplified when it is performed from within the operating system itself. *InsideTM* is also ***hooks*** exported functions of the operating system; these functions are categorized as various objects (files, registry, processes, or services), and each group is assigned to a separate analysis component, which provides information about the way the malware affects the objects. Work on ***TTAnalyze*** has continued under a new name, ***Anubis*** [57], and the name was later changed to ***LastLine***.

CWSandbox [58] is based on ***code injection*** of a trusted DLL into the analyzed process, which intercepts function calls by overwriting entries in the *Export Address Table* (EAT) and redirecting execution to ***CWSandbox***. This table is used by the Windows OS to map a function name to its address. Overwriting the EAT is possible, since the injected DLL is executed within the context of the analyzed process, giving it full control of all of the memory regions of the process. ***CWSandbox*** collects a vast amount of information (such as the name of the function called, its parameters, the system's state, etc.), which is then presented to the user. Research on ***CWSandbox*** has continued under the commercial name of ***ThreatAnalyzer***.

Capture [59] analyzes the state of the operating system using three monitors: the *file system monitor* (tracks read/write events on all hard drives), the *registry monitor* (tracks multiple registry events such as OpenKey, CreateKey, etc.), and the *process monitor* (tracks creation and termination of processes). All monitors provide additional data such as the process that triggered the event, full path(s), and timestamps. Using a kernel driver, ***Capture hooks*** kernel events that correspond with each of the above-mentioned monitors. The final result is a list of events triggered by the malware, with their timestamps and parameters.

MalTRAK [60] is a framework for tracking malware behavior and reversing its affects. It uses kernel mode components and ***hooks*** itself to key functions to track the malware's actions, while providing a mechanism for undoing these actions.

dAnubis [61] was developed to analyze kernel drivers and detect rootkits. It uses an in-guest component to monitor communication between the rootkit and the rest of the system. A trigger engine invokes various Windows API calls to reveal the presence of the rootkit (e.g., hidden processes or files).

7.2 Execution Control

Dynamic malware analysis should incorporate a mechanism to stop the malware execution once in a while and check the state of the malicious process and the OS. Execution control techniques include:

[38] https://docs.microsoft.com/en-us/windows/desktop/winmsg/about-hooks.

Debugging (also known as ***single stepping***) is a reliable analysis technique, originally developed to help programmers find errors in their code. Using the CPU's *trap flag* to generate an interrupt after each opcode instruction, a *debugger* can allow the malware to run only one opcode instruction before forcing a context switch back to the analyzing process, which can then examine the state of both the malware and the OS. After analyzing the operation, the *trap flag* is set again, and the malware continues its execution. This is a very resource-consuming technique due to the frequent context switches between the malware and the analyzing process. Another downside to ***debugging*** is that it is very easy to detect by malware (e.g., by using the *PUSHF* opcode instruction, which pushes the value of the *trap flag* to the stack). By checking if the debug flag is set to one, malware can detect if it is under analysis and hide its malicious activity.

Binary Instrumentation is a technique that adds analysis code to the original malware code during run-time. This is done by decompiling the original code until encountering some control transfer opcode instruction (such as *CALL, RET, JMP*), which signals the end of an *execution block*. Additional code is then "instrumented" into the block, introducing the desired analysis technique. The return address of the block is altered so it returns to the analysis tool, to allow execution of the next block. The resulting instrumented block is then executed. A new block is obtained by repeating the above process, starting at the last execution point.

Dynamic ***forward symbolic execution*** [62] is used to evaluate the malware execution using different inputs or environments. To bypass any *logic bomb(s)* used by the malware, *symbols* are created to replace input. For example, when the malware queries the OS about the current timestamp, no concrete value is return. Instead, the symbol created replaces it. At first, the symbol has no constraints and represents any possible value. Each condition tested against this symbol creates a new constraint (or branch). Keeping track of the input and its effects on the malware is primarily done using *data tainting* (Section 7.3). This technique can be used to map the malware code and the required conditions for executing each code section.

Based on the conditions tested by the malware, different code is executed depending on whether the conditions are met or not. ***Multiple path exploration*** [63, 64] is a mechanism used to force the execution of all of the malware's code, which is based on the concept of executing both paths of a selected condition. This method requires a way to store the system state before starting the first path (using a snapshot is one option) and analyzing the behavior of the malware. Once the execution is finished, the system is restored to the previous state, and the second path is explored. This technique has several limitations: this process cannot be applied for every condition (because of the increase in complexity), it might take a long time for the entire sample to be analyzed, and it requires a great deal of storage space to maintain all of the copies of the system's state.

DynamoRIO [65] is using a technique similar to ***binary instrumentation***, referred to as *run-time code manipulation*. It uses block caching to reduce the overhead required for interpretation. When encountering a control transfer opcode instruction at the end of an execution block, ***DynamoRIO*** checks if the target block is present in its blocks' cache. If so, then both blocks are linked, loaded, and executed, thereby reducing the number of context switches between processes. ***DynamoRIO*** is currently under development as an open-source project in *GitHub*.[39]

VAMPiRE (2005) [66] (a component within the *WiLDCAT* toolkit) introduced a technique called ***stealth breakpoints*** to *control execution*. By manipulating the attributes of memory pages ***VAMPiRE*** forces a page fault exception after each opcode instruction. Analysis is performed by installing a *page fault handler* (PFH), which analyzes the state of the malware and operating system. The PFH determines the type of breakpoint, analyzes the malware, and sets the next breakpoint before returning execution to the malware. Altering the attributes of memory pages and PFH

[39] https://github.com/DynamoRIO/dynamorio.

installation is possible by using a kernel component. ***SPiKE*** [67] is another *WiLDCAT* analysis component that relies on ***VAMPiRE***'s implementation to *control execution* of malware and uses ***binary instrumentation*** for malware analysis.

Cobra [68] (also part of the *WiLDCAT* toolkit) introduced the concept of *stealth localized executions*. An analyst can set specific parts of the code that require *fine-grained analysis* (see Section 7.3) by creating starting points (*overlay points*) and exit points (*release points*). When an overlay point is reached, ***Cobra*** uses ***binary instrumentation*** to execute code between these overlay points to provide *fine-grained analysis* for specific code segments.

MineSweeper [69] is a tool for analyzing a full-scale software using ***forward symbolic execution***. It includes a path selector mechanism that is used to choose the most promising path to explore next. By solving the constraints on each symbol ***MineSweeper*** detects when a particular branch is not reachable and removes it from the list of branches waiting to be explored. ***MineSweeper*** produces a graph representing the different behaviors the software can demonstrate and the conditions for each execution path. Although it cannot analyze malware by itself, it can be used to develop other analysis tools. ***MineSweeper*** was developed as a part of *TEMU* analysis platform (see Appendix).

Ether [70] is an analysis tool based on the *Xen* hypervisor without any in-guest components, thus preventing the malware from detecting the analysis tool from within the guest. To execute hidden ***debugging***, ***Ether*** hides the trap flag's state is from the guest OS by intercepting the *PUSHF* opcode instruction and fixing the flag's value before pushing it to the stack. ***Ether*** is also capable of handling cases in which the trap flag is intentionally set inside the guest by pushing the correct flag value to the stack. To prevent time-based detection, the system's tick counter is controlled to deceive the malware about the execution speed.

Arancino [71] is capable of detecting when a malware tries to detect the presence of ***binary instrumentation*** and applying the needed countermeasure. As research on malware analysis progresses, so does the sophistication of malware authors. Several techniques have been developed to detect and disrupt the process of ***binary instrumentation***. These techniques were surveyed as part of the development of ***Arancino***.

7.3 Flow Tracking

Fine-grained analysis is used for tracking the flow of information through the malware executed code (e.g., when a result of one function is used as a parameter to call another function).

Data Tainting [62] is a technique that "labels" information in its binary form. Interesting information (such as input from the user or data received over the network) is given a label (a *taint*) to indicate its origin. When the malware executes an opcode instruction that manipulates or processes tainted data, the taint is applied to any memory region affected. When tainted data reaches some predefined piece of code or memory (also called a *tainting sink*), the analysis process can retrace the flow of information. Tracking tainted data provides deep insight into the way the malware interacts with the operating system and the user. Note that the analysis complexity grows exponentially as more data flows into the system and more taints need to be tracked and applied simultaneously. ***Data tainting*** was originally used for intrusion detection to provide a mechanism for exploit detection. Tainting the data received from untrusted sources (such as input from a user) made it possible to detect *buffer overflow* occurrence (an indication for a possible exploit). Following the tainting path reveals where and how the input affected the system, and any changes made as a result of the overflow will be easily detected.

Other techniques like control ***flow tracking*** [72] or ***information flow tracking*** share many similar traits with ***data tainting*** and thus will not be covered here.

Vigilante [73] is an analysis tool implementing ***data tainting*** with ***binary instrumentation***. Originally developed to detect and stop the propagation of worms, ***Vigilante*** looks for the execution of tainted data originating from the network. As such, the taint source is the network, and the taint sink is reached when the instruction pointer (IP) points to tainted data, meaning that some untrusted code that came over the network is being executed.

Panorama [74] started as a *TEMU* plugin for ***taint analysis*** of various I/O devices, including a hard drive, keyboard, or mouse. It later became an independent platform. The output of ***Panorama*** is provided in the form of a graph, which allows the user to track the flow of data between processes and memory regions.

Dytan [75] is an extension to the *Pin* instrumentation system, providing an easy to use API for ***data tainting***. It was developed with a flexible design that allows the user to configure various components such as taint sources, taint sinks, data flow tracker, and the control flow tracker. ***Dytan*** can be configured to track explicit and implicit flows of information. In addition, its functionality can be extended by using callback functions that implement additional taint sources, labels, propagation, and sinks.

TQana [76] is a framework built on top of *QEMU* for analyzing and detecting malicious browser extensions installed on Internet Explorer. It uses ***data tainting*** with two taint sources: (1) all of the URL strings of the pages that a user visits and (2) the information that the browser receives in response to its requests. The taint sinks are the file system, registry, and network. When tainted data is written to a file or sent over the network, the analyzed sample becomes suspected of being a *spyware*.

7.4 Tracing

Gathering information left after the execution of some code is called tracing. Network connections and the memory allocated to the malware leave traces of the malware's behavior. Analyzing these traces can offer insights about the malware without using an in-guest components.

Volatile memory analysis—Analyzing the effects of malware from memory dump files (see Section 6.5) requires an understanding of how the operating system keeps track of processes, files, users, and configurations. All of these data structures exist in the memory dump in binary form. Rebuilding the original operating system state from a memory dump is called *bridging the semantic gap*, and it can be done by reverse engineering the OS's structure or by using a third-party tool (e.g., *Volatility*).

A memory dump provides a better perspective about the state of the system at a given point in time than any other analysis layout. ***Volatile memory analysis*** is done on a static copy of the system without querying the operating about its internal structures of processes or drivers. Once the memory dump is acquired, analysis can be performed safely and without fear that the malware might tamper with the result of the analysis. Apart from user mode code, the memory dump also contains the kernel code and all of its structures, making ***volatile memory analysis*** the leading choice for analyzing rootkits [77], malicious firmware [78], and fileless malware.

Network Tracing—Since most of the time an Internet connection is required for malware to perform its actions, the exact nature of the malware might not be revealed without Internet access. However, granting malware full Internet access is sometimes not desirable or possible. Limiting network access by malware and analyzing the network connections can reveal the malware's C&C and commands received from it. Networking traces left by the malware are useful for understanding the communication patterns it presents.

HookFinder [79], another part of the *TEMU* implementation, is designed to detect and analyze malicious hooks by analyzing volatile memory. The information found in the stack is translated to create a hook graph, which helps in identifying the hooking chain. ***HookFinder*** then labels

memory segments that belong to the malware as *taint sinks*. To verify that a hook was in fact installed by malware, ***HookFinder*** invokes various function calls and ***tracks the control flow*** by checking the *instruction pointer* (IP). When the IP points to a tainted memory, the malicious hook is revealed and verified.

LiveDM [80] utilizes *QEMU* to analyze the allocation of new memory regions in the kernel. By hooking several memory allocation functions implemented by the operating system, it tracks where the malware installs itself and performs static analysis on the malware's binary code. Using the emulator's control over the guest OS, ***LiveDM*** is able to obtain the guest OS's volatile memory and analyze the infected kernel.

TrumanBox [81] is an analysis tool that implements ***network tracing***. ***TrumanBox*** is a network bridge with emulation capabilities, meaning that it uses two network interface cards (NICs): one connected to the analysis OS and the other connected to the Internet. When the malware tries to open a network connection, ***TrumanBox*** either forwards the request to its target or emulates the response itself. Various off the shelf networking services are implemented (e.g., DNS, DHCP, IRC, FTP, and SMTP).

Vis [82] is a tool designed to perform ***volatile memory acquisition*** with minimal changes to the operating system to avoid detection by malware. ***Vis*** installs a trusted hypervisor before booting the OS, giving ***Vis*** full control of the native operating system. It then uses these privileges to pause what is now the guest OS and dump the volatile memory to a file. This file can then be analyzed using a variety of volatile memory analysis tools.

Actaeon [83], a plugin to the *Volatility* framework to analyze volatile memory containing *type 1* and/or *type 2 hypervisors*. When using a hardware device to perform *volatile memory acquisition*, the memory dump might contain a hypervisor. The analysis of such memory dumps must consider the unique memory objects of the hypervisor. ***Actaeon*** can analyze malicious hypervisor (*VMBR*) and malware using *nested virtualization* (see Section 4.1) by detecting the unique data structures shared by all hypervisors, without relying on signature-based detection.

MASHKA [77] is a tool for ***volatile memory acquisition and analysis*** that focuses on the detection of stealth rootkits. In this case, the memory is dumped by traversing the memory page table, enabling it to reach pages that were swapped out to the hard drive by the operating system and are not currently present on RAM. Most rootkits hide from the user by removing the malicious process from the active processes list. By comparing the *EPROCESS* structure of all processes, ***MASHKA*** can identify the anomalous process the rootkit is hiding. This, along with the analysis of other kernel objects, can help with rootkit detection.

AMAL [84] analyzes malware in a guest VM using *VMware*'s implementation for virtual machines. ***AMAL*** has a configuration feature enabling it to set different properties of the VM (e.g., choosing the operating system, modification of registry keys, installation of specific software) before executing the malware. Each analysis is executed for a configurable amount of time, and a snapshot is taken (containing the RAM and file system of the VM). ***AMAL*** runs several unspecified tools to track the file system, registry, network, and metadata. ***AMAL*** also performs ***volatile memory analysis*** by running YARA signatures on the memory to identify malware of interest.

To bypass anti-forensic methods developed by malware authors, Cheng et al. presented an ***unnamed forensic framework*** [85] that utilizes the hypervisor technology to perform trusted ***volatile memory acquisition and analysis***. This is done by installing a trusted hypervisor and inspecting the guest OS's volatile memory. This framework can be used to detect rootkits by comparing the processes list with the RAM's binary representation.

CIASandbox [86] is an analysis platform based on the *Xen* hypervisor. Malware is executed in the guest OS and inspected from the outside by the hypervisor. Information about system calls made is obtained through the traces left in the guest VM's memory and stack. ***VTCSandbox*** [87]

extends ***CIASandbox*** to utilize *Xen*'s *virtual time controller* (VTC) to manipulate time inside the guest VM and accelerate analysis. Since most analysis platforms execute suspicious malware for a limited period of time before terminating the analysis, malware authors include delaying mechanisms (such as *sleep* commands) to evade analysis. The solution presented by the developers of ***VTCSandbox*** manipulates the tick counter to speed up execution inside the VM, which helps skip sleep time or other delays malware use to suspend their operation.

Nissim et al. [48] and Cohen et al. [49] presented ***unnamed*** machine-learning-based methodology for ***volatile memory analysis*** of virtual servers'. *Volatile memory acquisition* was performed by a *hypervisor*, and various features were extracted from the memory dumps. These features were then leveraged by machine-learning algorithms to induce a malware detection model (see Section 12).

7.5 Side-channel Analysis

The analysis techniques presented so far rely on extracting data from the operating system, volatile memory, or the emulated machine's state. However, any type of computational device can be targeted by malware. Such devices include PCI cards, IoT devices, hard drives, medical devices, and so on. Analysis and detection of malware running on such devices is difficult, since most of the time these devices do not contain an operating system that can support the traditional analysis techniques. Instead of tracking the behavior of the system from the OS's perspective (or from the binary level), it is possible to analyze the behavior of physical components by their power consumption, EM emissions, or internal CPU events (see Section 6.6).

The data acquired is divided into "normal behavior" and "infected behavior." Using statistical methods and machine-learning algorithms, the detection of a deviation from normal behavior might indicate abnormal CPU behavior (e.g., the presence of a cryptominer or rootkit). ***Side-channel analysis*** cannot provide in depth knowledge about the internal events of the operating system, the network, or the files being modified. No report is provided to the user who simply receives the final verdict (known as malicious or not).

WattsUpDoc [13] is an analysis tool to perform *side-channel analysis* of medical devices using external equipment. It demonstrated that *side-channel analysis* can be used for analyzing devices that don't have an operating system and without loading any code to the analyzed device.

NumChecker [14] is a platform for rootkit analysis and detection. It uses a VM to measure the *hardware performance counters* (HPCs) from the host. Using an in-guest component to signal when a *system call* has been triggered, ***NumChecker*** checks the HPC before the *system call* is started and checks it again when it returns. Thus, the HPC tracks only the performance of *system calls*. By analyzing a clean system (offline phase), ***NumChecker*** creates a normal execution pattern and compares it to the HPCs measured while executing unknown binaries. If a rootkit has been successfully installed, then it should create a different execution pattern than normal, which can be detected by ***NumChecker***.

Demme et al. [15] developed an ***unnamed*** malware detection platform for the detection of Linux rootkits and Android malware using hardware performance counters. Although the paper does not elaborate on the framework itself, it does provide an in-depth description of their use of machine-learning algorithms to detect the infected operating systems.

Data gathered from hardware counters can be used to train an anomaly detector. Tang et al. [16] have shown how a profile of a "clean" state can be created using unsupervised learning algorithms and deviations from this profile can be used as a malware detector (***unnamed tool***). The paper describes the experimental setup and the framework's design, although insufficient information is provided about the experiments and dataset used.

Using HPCs to detect malware has been tested by various researchers with inconclusive results. One study [88] showed that while it is possible to use HPCs to detect ROP (*return-oriented programming*) execution, HPCs are less effective for malware detection. ROP is a common technique used by malware to counter anti-exploit mechanisms, which is based on reusing compiled code that was previously loaded into the victim's RAM. The low-level behavior of ROP is identifiable by its high volume of RET opcode instructions (which signal a return from procedures) without CALL opcode instruction (which signals procedure calls) [89].

Since the HPCs provide insight regarding the low-level behavior of the CPU, they offer a strong indication whether an ROP is being executed or not. However, most malicious code is not executed with a ROP mechanism, so the question regarding whether or not HPC events can provide significant evidence for the presence of malware (other than ROP) remains open. In our current survey, we were unable to identify prior research that supports or rebuts the abovementioned question, and we hope that future research will confirm or contradict this statement.

In a recent paper [90] that reviewed various publications about malware detection using HPCs the authors identified five major drawbacks when using HPCs to detect malware and conducted experiments to test the correlation between the HPC measurements and malware behavior. Their paper also includes a comparison between different algorithms, and best result of ~81% F1 score demonstrates that the correlation between HPC measurements (low-level events) and malicious software (high-level code) is strong but not strong enough to create a reliable malware detection mechanism.

An additional review paper [88] surveyed over 50 publications that cover the topic of HPC. The paper also includes an experiment comparing ROP detection and malware detection. In this paper, the authors demonstrated that while the low-level behavior of ROP can be detected using HPCs, it is not the case when considering malware detection.

The two comparison papers mentioned above [88, 90] point out the weak connection between HPC measurements and the high-level code executed by malware, but the same conclusion can be drawn by side-channel analysis using external hardware. The conclusion is that side-channel analysis is only as strong as the statistical correlation between two separate views: high-level code and low-level hardware. Although this correlation does exist, it is not perfect. Future researchers and security professionals should keep this limitation in mind.

8 MAPPING TECHNIQUES TO LAYOUTS

Figure 12 presents a mapping of the analysis layouts (Section 6) to the techniques (Section 7) that can be implemented on them. The mapping enables us to draw some meaningful insights; for example, due to the inherent behavior of ***function call analysis, execution control***, and ***flow tracking***, they can only be implemented with an in-guest component or on bare metal. ***Volatile memory analysis*** is applied to memory dumps, and ***side-channel analysis*** is applied to power consumption traces of the physical device. The last two techniques do not require any in-guest components, which reduces the risk of detection or subversion by malware. ***Side-channel analysis*** require information extracted from the physical device, which is why this technique cannot be applied when using an *emulator*. We hope that researchers will find this figure helpful when designing new analysis techniques or layouts.

9 MALWARE BEHAVIOR VS. ANALYSIS FRAMEWORKS

The battle between malware and analysis tools is at type of arms race. Attackers are continuously developing new methods to evade and detect analysis frameworks, while the ability of analysis frameworks and tools tasked with detecting malware keeps improving. Figure 13 was created to help readers understand this arms race and the different methods used by attackers and analysts.

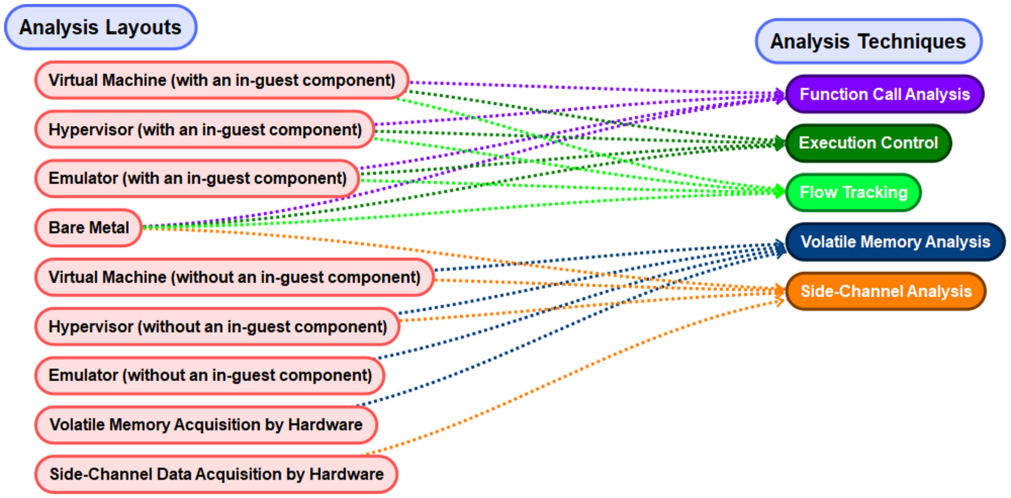

Fig. 12. Mapping of layouts and techniques.

On the left side of the table, we list malware behaviors (Section 4), while the analysis layouts (Section 6) and the various analysis techniques (Section 7) are listed on the top. Note that static analysis and unpacking were included in the list of techniques, because they provide some insights regarding the malware prior to its execution that can be useful for the dynamic analysis phase.

The figure presents in red the behaviors used by malware to evade or infect the analysis framework. The green represents the analysis layouts and techniques used to track the malware's operations or prevent the malware from infecting the system. The blue squares are particularly interesting, as they represent the cases where malware and analysis are engaged in an arms race. In such cases, the ability of the analysis to detect the malware is dependent on how well the analysis technique is implemented and the level of sophistication employed by the malware. Figure 13 is followed by two subsections in which we provide some insights from this figure.

9.1 Malware Behavior—Insights

Rootkits and VMBR—Privilege escalation allows malware to infect bare metal machines and evade control execution techniques. Such privileged malware can be analyzed using a *guest-host model*, *volatile memory forensics*, or *side-channel analysis*. A VMBR, unlike a kernel rootkit, can take over virtual machines if a malicious hypervisor has successfully been installed.

Hardware infection—By installing malicious firmware on hardware devices, malware can infect almost every analysis layout. Such malware can be analyzed by *emulation* or *side-channel analysis*. Like VMBRs and rootkits, hardware infection allows the malware to take over a system; it also provides the added bonus of resiliency to *volatile memory acquisition*.

Detecting the analysis framework—When implemented by hardware, *volatile memory acquisition* and side-channel analysis are the only trusted layouts that cannot be detected by malware. While *control execution* techniques can be detected by malware, *multiple path exploration* can force malware to execute its malicious code thereby bypassing any detection mechanism used my malware to detect the analysis.

Other logic bombs—*Multiple path exploration* can bypass logic bombs in a generic way without the limitation of having to reproduce the correct conditions for the code to be invoked "naturally."

Low-level actions—Malware evade *function call analysis* by using functions that might not be monitored by analysis tools. Analysis on the binary level (using *binary instrumentation* and *data*

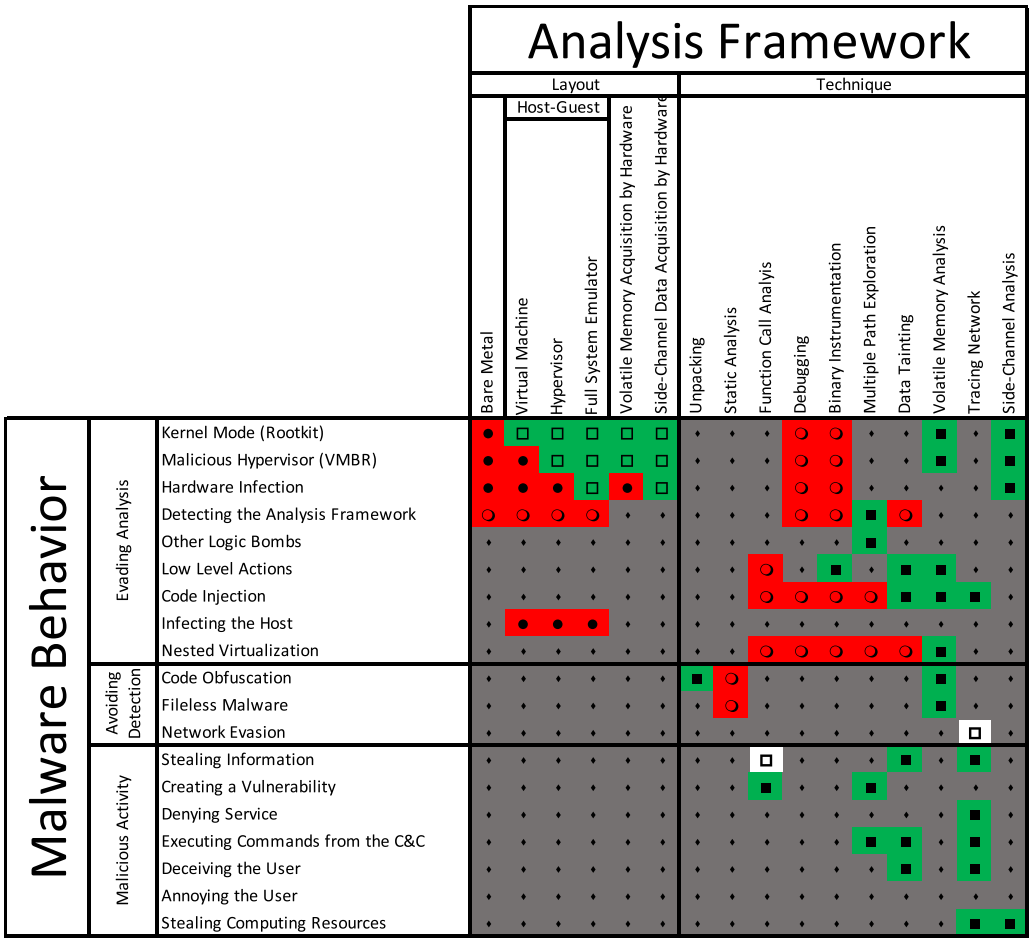

Fig. 13. Summarization of malware behaviors and their correlation with analysis layout and techniques.
- Malware behavior designed to infect the analysis framework
- Malware behavior designed to evade analysis
- Analysis techniques designed to collect data about malware behavior
- Analysis layouts designed to prevent malware infection
- Cases where malware and analysis technique are engaged in an arms race

tainting) reveals the malware behavior, and *volatile memory analysis* can be used to detect the traces of malware behavior.

Code injection—Useful for evading analysis when the analysis technique is focused on a single process (common when implementing *function call analysis* or *execution control*). *Data tainting* reveals the malicious code added to the targeted process, *network tracing* reveals if any communication is made by the injected code, and *volatile memory analysis* provides the actual binary code loaded into the memory space of the targeted process.

Infecting the host—When a vulnerability in the guest-host implemented is found, malware could try and infect the host. If the malware is successful, then the analysis framework is no longer trusted, since the malware is running outside the analysis tool's scope. Patching vulnerabilities as soon as they are discovered provides mitigation of this behavior.

Nested virtualization—Running malicious code inside a VM hides the internal working of the malware behind the semantic gap [91]. *Function call analysis* and *control execution* techniques are useless in this case, and the process be analyzed only with *volatile memory analysis*.

Code obfuscation—Malware hide signatures to evade static analysis methods. *Unpacking* the file before starting the analysis might be useful to reveal the original binary code; *volatile memory analysis* reveals the actual executed code and eliminates the need for unpacking.

Fileless malware—Malware residing only in memory cannot be *statically analyzed* (there is no file to analyze), only *volatile memory analysis* can be used.

Network evasion—Malware authors use various technique to evade network detection. Analyzing the networking behavior requires a solution to each of these evasion techniques, which in turns leads malware author to develop a new evasion technique. This is an example where malware and analysis are engaged in an arms race.

Stealing information—Malware collect information about the user's behavior (and the infected system in general) by installing various hooks (e.g., keylogger malware). Analysis tool developers who implement hooks as a function call analysis mechanism need to verify that their tool does not interfere with the malware and that the malicious behavior is being analyzed correctly.

Creating a vulnerability—This behavior can be captured by tracking the various functions invoked to install the vulnerability (e.g., new user creation, configuration changes). Other types of vulnerable code (such as various backdoors) are only invoked when specific events happen. *Multiple path exploration* can reveal these cases during the analysis process.

Denying service—Analyzing the behavior of malware performing a denial-of-service attack (including the bot behavior used for DDoS attacks) is possible by *tracing the network*. Other types of service denial (using lockers and ransomware, or damaging hardware) are not analyzed in a specific way that merits noting in this table.

Executing commands from the C&C—Capturing this behavior is possible by allowing the malware to communicate with the C&C and *tracing network activity*; by trying to invoke code using *multiple path exploration*; or by following the propagation of input throughout the malware's code using *tainting*.

Deceiving the user—Phishing attacks can be analyzed by *tracing the network*, while malware impersonation (of another user or process) can be analyzed by *data tainting*.

Annoying the user—This type of behavior is not well-defined; thus, none of the analysis techniques presented in this survey is designed to mitigate this malicious behavior.

Stealing Computing resources—Analysis of resource usage should be performed at the hardware level. Thus, *side-channel analysis* is the only possible technique. *Tracing the network* can reveal communication between the malware and its operator (e.g., the requests made to the mining pool by cryptominers).

9.2 Analysis Layouts—Insights

Bare metal—This is the least-trusted analysis layout, because the analysis tool can run at kernel mode at best and cannot be trusted for analyzing malware with privileges higher than *user mode*.

Virtual machines, hypervisors, and emulators—These layouts offer a guest-host separation that is more trusted than bare metal, and also allows restoring mechanisms, but they also require either *bridging the semantic gap*, or using an in-guest component (which is vulnerable to detection or subversion by malware). These layouts are also vulnerable to implementation flaws, and in rare cases, they can allow malware to escape the guest and infect the host.

Volatile memory acquisition by hardware—This layout has the advantage of a trusted mechanism for memory acquisition without an in-guest component. The downside to this layout is that malware behavior can only be observed by the traces left between one memory dump and the next.

Side-channel data acquisition by hardware—Observing behavior from the hardware level is the most trusted layout of all. Acquisition and analysis cannot be detected or subverted, because it does not compete with the malware for the same resources. However, the analysis cannot reveal the internal workings of malware.

To summarize the insights from Figure 13, the best way for malware to avoid analysis is to escape the analysis environment (by **infecting the host OS**, by **code injection**, or by **nested virtualization**). Achieving this requires in depth knowledge of the analysis platform to find flaws in its design and implementation. Gaining higher privileges is also very effective. **Rootkits, VM-BRs**, and **hardware infection** can also overcome weak analysis tools. Malware can also evade analysis tools by **detecting the analysis framework** and refraining from executing malicious code.

Achieving a truly trusted analysis mechanism is possible by giving up on in depth analysis and making do with less informative analysis. A perfect analysis platform does not exist, and some concessions must be made. In our opinion, ***volatile memory forensics*** offer a good balance between analysis granularity and protections from infection or detection: Analyzing a static memory dump prevents malware from altering the information or detecting the analysis itself, while providing a complete overview of the system's state at a given point in time. However, the analysis is not continuous and depends on the frequency at which memory dumps are taken. We believe that these pros outweigh the cons.

10 COMPREHENSIVE COMPARISON OF THE STUDIES

In this section, we compare the dynamic malware analysis studies conducted over the years and discuss trends and parameters associated with such analysis. Performing this comparison enabled us to discover some important insights that we share with the reader as well.

10.1 Comparison Based on Functionality and Practical Aspects

Figure 14 presents one of our core contributions in this article and provides a thorough overview and comparison of the academic tools surveyed in this article. We base our comparison on several key aspects: (1) the tool's **Relevance** (its impact and contribution to the scientific community based on citation numbers and whether or not it is open-source), (2) the **Versatility** of the analysis the tool provides, (3) the analysis **Layout** (see Section 6) used to implement the tool, (4) the tool's **Limitations** (pre-analysis requirements, additional software it depends on, special required hardware), and (5) the **Output** provided by tool. Note that the tools are grouped by analysis technique, and within each technique they are ordered by their publication year. A detailed explanation about Figure 14 follows:

Relevance—Interestingly, only 14% of the studies have chosen to open-source their code. We believe that open-sourcing should become today's standard for academic research, as it encourages increased evaluation and allows others to contribute and expand upon previous work. The leading tool implementing each analysis technique is clearly seen by comparing the average citations, and the figure demonstrates that the papers on ***CWSandbox, Ether, Panorama, memory acquisition using a hypervisor***, and ***performance counters*** have had more significant impact than other papers.

Versatility—While most tools are focused on binary and document analysis, less than half of the tools are capable of analyzing kernel code, and only two offer plugins for extending functionality.

Layout—We include the *analysis layout* implemented by the tool. For those tools that implemented *virtual machine, hypervisor*, or *emulator*, we also included information regarding the existence of an in-guest component.

Limitations—Some of the tools have limitations, which are described below:

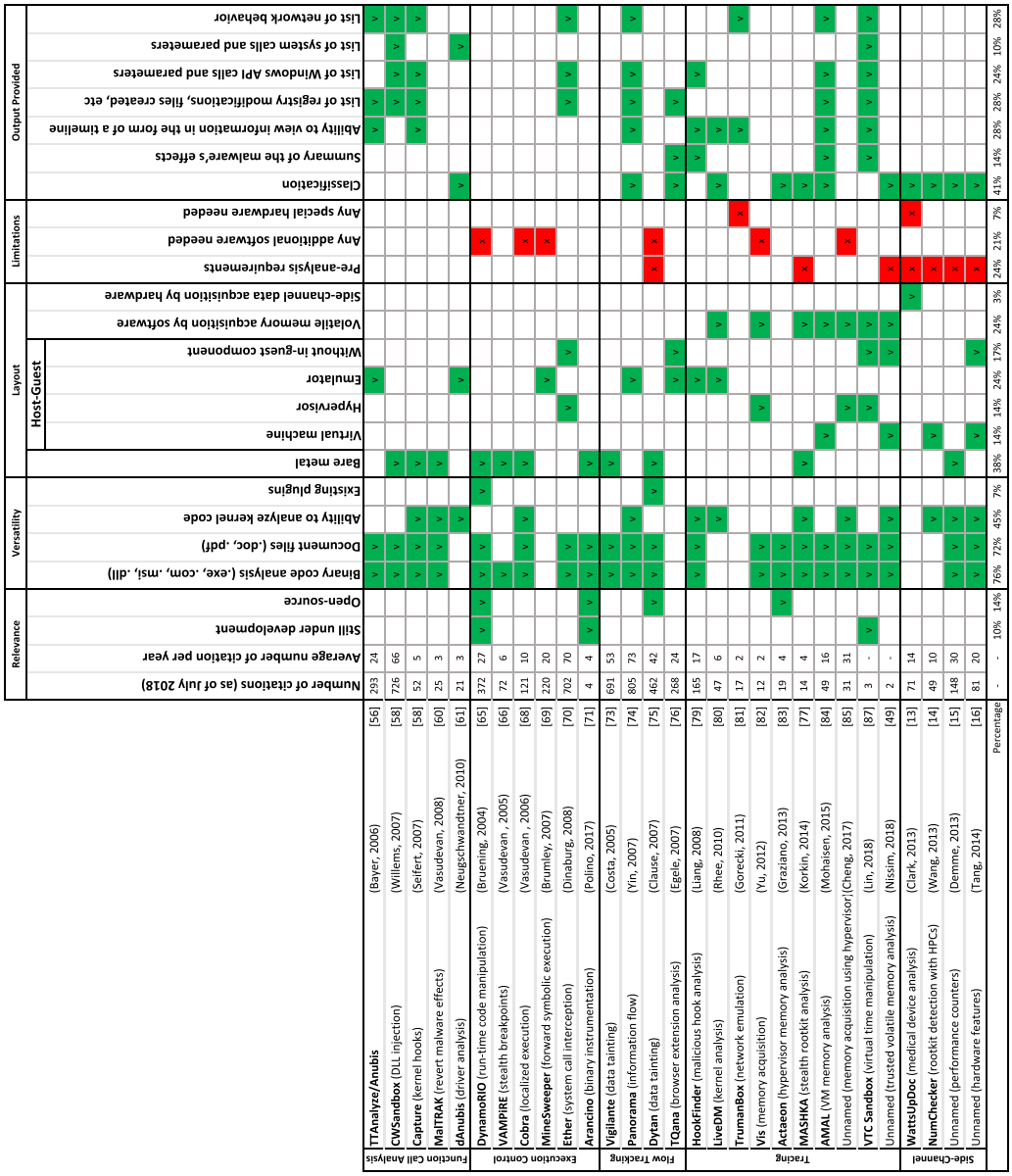

Fig. 14. A comparison based on functionality and practical aspects (limitations are marked in red for convenience).

(1) Pre-analysis requirements: ***Dytan*** requires that tainting sources and sinks be defined before starting the analysis. ***MASHKA***'s implementation might interfere with various DMA devices, therefore the memory addresses of such devices must be added to an ignore list before starting the analysis. For the ***unnamed memory analysis tool*** and all *side-channel analysis* tools a baseline of clean behavior must be created before executing the malware.

(2) Additional software: ***DynamoRIO, Cobra, MineSweeper***, and ***Dytan*** were not developed specifically for malware analysis but instead provide a platform for future development of malware

Table 1. Footnotes for Figure 15

Abbr.	Meaning
x	Appears when the information was not included in the paper
*	Executables, documents, rootkits, browser extensions, hypervisors, medical devices, websites (bold letters represent the values in the table)
**	Malicious file percentage
***	Accuracy and precision (when provided)—the best results presented in the paper
****	Demonstrated superiority over these tools: (1) Evaluation has shown it was superior to Renovo [92], PolyUnpack [93], Anubis, and Norman Sandbox[40] (2) Evaluation has shown it was superior to Anubis (3) Evaluation has shown it was superior to Ether

analysis tools. Volatile memory acquisition tools require a 3rd party software (e.g., *Volatility*) to analyze the memory dump acquired.

(3) Special hardware: ***TrumanBox*** requires customized hardware to bridge the client and the Internet, while ***WattsUpDoc*** uses measuring equipment to trace the power consumption of the analyzed devices.

Output Provided—These columns provide a summary of the output report, showing that most of these tools did not provide comprehensive output to the user. If the output report is not informative enough to allow an analyst to determine what a malware actually does, then it is not useful. An output report should include as much information as possible about malware behavior.

At the bottom of Figure 14, we provide the percentage of the tools that comply with each of the aspects we used to compare the tools. For example, 76% of the tools analyze binary files and non-executable files (with the exception of ***VAMPiRE***, which is designed for creating stealth breakpoints and thus is limited to binary files). The remaining 24% focus on analysis of specific components such as kernel drivers, network behavior, medical devices, and more. Only 41% of the tools provide a classification regarding the analyzed file, and 31% include network behavior in their output.

It is interesting to see the progression over time. While early research was focused on *function call analysis, execution control*, and *flow tracking*, over the last 10 years researchers have begun to focus more on analyzing the traces left by malware in the analyzed environment (host, device, network, etc.). This shift in focus is a result of trying to avoid using in-guest components for malware analysis, which led to development of *volatile memory forensics* and *side-channel analysis*.

10.2 Comparison Based on Research Evaluation Measurements

Figure 15 provides a comparison of the surveyed academic tools based on the evaluation measurements included in their respective papers. Such evaluation measurements include details about the dataset used for evaluating the tool (type and size), number of malicious or benign samples included (the mixture), true-positive rate (TPR), false-positive rate (FPR), and so on. Table 1 provides that footnotes description for Figure 15.

It is notable that some of the papers surveyed in this article were published without including sufficient information describing their tests and datasets. The malicious file percentage (MFP) column shows that most of the time, the papers that included an evaluation section only tested

[40] http://download01.norman.no/product_sheets/eng/SandBox_analyzer.pdf.

			Authors' Evaluation								
			Dataset *	Malicious samples	Benign samples	MFP **	TPR	ACC ***	FPR	PRE ***	Superiority ****
Function Call Analysis	**TTAnalyze/Anubis**	(Bayer, 2006) [56]	E	8	0	100%	×		×		
	CWSandbox (DLL injection)	(Willems, 2007) [58]	E	6148	0	100%	×		×		
	Capture (kernel hooks)	(Seifert, 2007) [58]	D	1	0	100%	×		×		
	MalTRAK (revert malware effects)	(Vasudevan, 2008) [60]	E	8	0	100%	×		×		
	dAnubis (driver analysis)	(Neugschwandtner, 2010) [61]	R	463	0	100%	×		×		
Execution Control	**DynamoRIO** (run-time code manipulation)	(Bruening, 2004) [65]	E	0	30	0%	×		×		
	VAMPiRE (stealth breakpoints)	(Vasudevan , 2005) [66]	E	1	0	100%	×		×		
	Cobra (localized execution)	(Vasudevan , 2006) [68]	E	1	0	100%	100%		×		
	MineSweeper (forward symbolic execution)	(Brumley, 2007) [69]	×	×	×	×	×		×		
	Ether (system call interception)	(Dinaburg, 2008) [70]	E	25118	0	100%	×		×		(1)
	Arancino (binary instrumentation)	(Polino, 2017) [71]	E	7006	0	100%	×		×		
Flow Tracking	**Vigilante** (data tainting)	(Costa, 2005) [73]	E	3	0	100%	100%		×		
	Panorama (information flow)	(Yin, 2007) [74]	E	42	56	43%	100%		3%		
	Dytan (data tainting)	(Clause, 2007) [75]	×	×	×	×	×		×		
	TQana (browser extension analysis)	(Egele, 2007) [76]	B	21	14	60%	100%		3%		
Tracing	**HookFinder** (malicious hook analysis)	(Liang, 2008) [79]	E	8	0	100%	×		×		
	LiveDM (kernel analysis)	(Rhee, 2010) [80]	R	10	0	100%	×		×		
	TrumanBox (network emulation)	(Gorecki, 2011) [81]	E	300	0	100%	×		×		(2)
	Vis (memory acquisition)	(Yu, 2012) [82]	×	×	×	×	×		×		
	Actaeon (hypervisor memory analysis)	(Graziano, 2013) [83]	H	0	5	0%	×		×		
	MASHKA (stealth rootkit analysis)	(Korkin, 2014) [77]	×	×	×	×	×		×		
	AMAL (VM memory analysis)	(Mohaisen, 2015) [84]	E	115157	0	100%		99.2%		98.9%	
	Unnamed (memory acquisition using hypervisor)	(Cheng, 2017) [85]	R	1	0	100%	×		×		
	VTC Sandbox (virtual time manipulation)	(Lin, 2018) [87]	E	2788	1204	70%		98.6%		65.6%	(3)
	Unnamed (trusted volatile memory analysis)	(Nissim, 2018) [49]	E	10	6	63%	97.9%		0%		
Side-Channel	**WattsUpDoc** (medical device analysis)	(Clark, 2013) [13]	M	4115	861	83%		99%		100%	
	NumChecker (rootkit detection with HPCs)	(Wang, 2013) [14]	R	8	0	100%	100%		×		
	Unnamed (performance counters)	(Demme, 2013) [15]	E	503	210	71%	83%		10%		
	Unnamed (hardware features)	(Tang, 2014) [16]	E, D, W	Unspecified	~820	×	100%		10%		

Fig. 15. Comparison of the surveyed academic tools.

their implementation on malicious files, without including benign software as well (to determine if the tool produces any false positives); this fact leaves their evaluation incomplete and questionable.

11 DYNAMIC MALWARE ANALYSIS STEPS

Malware analysis is a structured process. The analysis should start with removing any packing wrapper applied to the binary code to hide it from the analysis tool (a process called unpacking

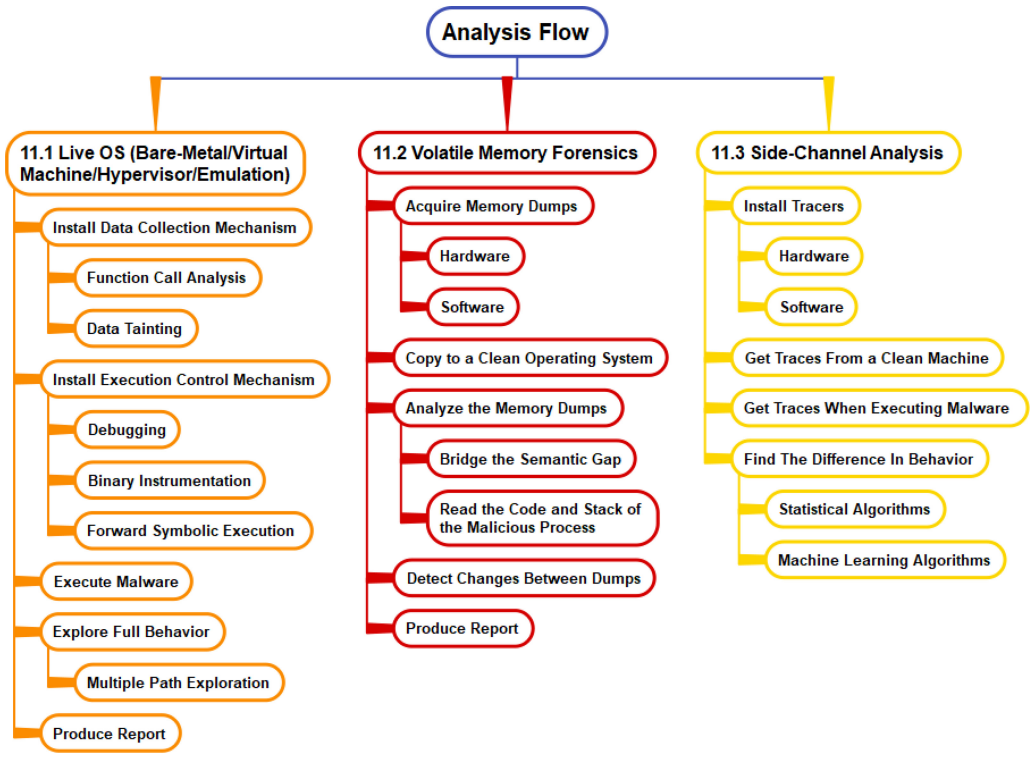

Fig. 16. Dynamic malware analysis steps (for each layout).

[92, 93, 94]). Once the unpacked file is obtained, a *static analysis* method can be applied to gain some insight about the file. If the file matches a known malware's signature, then the analysis process might be skipped completely, so static analysis is a basic step that can reduce the necessity for further analysis. Based on the analysis layout selected for the task, further steps must be taken. Figure 16 presents a summary of the steps required for each analysis layout, grouped by their functionality. A detailed explanation follows.

11.1 Malware analysis with Live OS Steps

Analysis performed on bare metal, virtual machine, hypervisor, or emulator can be considered live analysis ('live' in the sense that the operating system is running during the entire analysis process, and the data is obtained by software).

The first step of analysis on a "live" OS is ***installing a mechanism to collect data***. This includes using the function call analysis or data tainting techniques. Next, a ***mechanism to control execution is installed***. Such control includes the ability to stop the analyzed process at any point and inspect various aspects of the system. *Debugging* and *binary instrumentation* are common practices to control malware execution. *Forward symbolic execution* can be used for mapping different execution branches.

Once those mechanisms are in place, the ***malware is executed*** for some time. The duration of execution depends on the available time for analysis, and the desired extent of analysis.

Since the malicious code might only be executed when a set of conditions are met, the analysis framework could include a mechanism to ***explore the full behavior of the malware*** to reveal

its maliciousness. To combat the different l*ogic bombs* employed by malware, multiple execution branches can be explored.

After the sample has been executed and analyzed, the analysis tool needs to ***produce a report***. If the malware has stolen the user's password, for example, then the analysis report should indicate how it occurred (i.e., identifying which functions were called/hooked, the C&C address, whether the password stolen from the RAM or extracted through a phishing attack, etc.)

11.2 Malware Analysis with Volatile Memory Forensic Steps

Sometimes it is not possible perform analysis on a live OS. For example, when a server is suspected of being infected by malware but cannot be shut down or new software installation is prohibited by the administrator. In such cases, analysis can be performed on a volatile memory dumps.

Memory acquisition can be handled by hardware or software (see Section 6.5). Some implementations require that the acquisition mechanism be installed before booting the computer, while others support acquisition from running machines without rebooting (e.g., *CaptureGUARD*). Once the device or software is installed, the memory can be copied to a file (a binary representation of the OS and its processes) and saved on the host or external storage. During the dumping process special care should be given to memory pages that were swapped out to the hard drive and are not present in RAM. These pages need to be swapped in from the hard drive to be included in the dump file. The dumping process can be done in intervals to provide insights about changes made by the malware and its behavior over time. Although analyzing the dump file(s) can take place on the same machine, it is not recommended. Instead, the file should be ***copied to a clean machine*** that was not used to run live malware.

Analyzing memory dump files requires *bridging the semantic gap* (i.e., the internal structure of the kernel and process representations should be extracted from the file, including the stack and binary code). Observing the malware's stack and code over time exposes its function calls, variable values, network communication, and more.

Analyzing memory dumps acquired over time can reveal malware behavior by ***detecting the changes between dumps***. Such changes can be newly created processes, open files, function calls found in the stack, networking activities, and so on. Keep in mind that because the dump files represent the machine state at a specific time, they are "frozen" in time. If the time interval between dumps is too large, then information might be lost. For example, a function call (represented in the stack) will be available for a short period of time until the function returns and the stack memory is overwritten.

11.3 Malware Analysis with Side-channel Data Steps

Side-channel analysis starts with ***installing tracers*** for data acquisition, either by hardware (e.g., external measuring devices) or software (e.g., *hardware performance counters*). Since the CPU's state changes very rapidly, the traces should be obtained at high granularity, either by frequently querying the HPCs or recording the measuring device's output.

Traces extracted from a clean machine are used to create a baseline for normal behavior, while ***traces extracted during malware execution*** are compared to the baseline. When malware is executed, the system behavior is expected to change, and so does the measurements are expected to change. The recorded information from each execution (both malicious and benign) can be copied and analyzed statically on a separate machine. The recording does not represent any internal content of the operating system, so insights are gained by ***finding differences in behavior*** between the baseline to malware execution, usually by statistics or machine-learning algorithms.

Table 2. Prominent Studies that Use Machine-learning Algorithms for the Task of Malware Detection, Classification, and Analysis

Function Call Analysis	**Analysis of machine-learning techniques used in behavior-based malware detection** [20]	Firdausi, 2010	Detection
	Using Anubis (previously referred to as TTAnalyze) to extract an XML report file. The dataset consisted of 220 unique malware samples and 250 benign samples. All of the reports were used to create term frequencies as features. The J48 model achieved the best results (96% TPR and 2.4% FRP).		
	Automatic analysis of malware behavior using machine learning [19]	Rieck, 2011	Clustering
	CWSandbox was used to extract system calls from 3,133 malware samples. N-grams were created from system call sequences, and a clustering algorithm was developed to group similar malware. The authors tested their framework on 33,698 unknown malware samples and achieved an F-measure of 99.7% (FPR and TPR scores were not reported in the paper).		
	Malware classification with recurrent networks [95]	Pascanu, 2015	Classification
	Using a bidirectional recurrent neural network, the authors of this paper showed that neural networks can classify malware based on observed system calls. 114 distinct system calls were extracted from 250,000 malware samples. Their best model achieved ~82% TPR and ~0.5% FPR.		
Flow Tracking	**A machine-learning approach for classifying and categorizing android sources and sinks** [96]	Rasthofer, 2014	Automating Tainting
	The authors of this paper demonstrated a technique used to classify Android API functions into taint sources and sinks. The purpose of these sources and sinks was to detect malicious behavior in Android apps. After analyzing 11,000 samples, their best model obtained 92.3% precision and recall.		
Tracing	**Adaptive detection of covert communication in HTTP requests** [45]	Schwenk, 2011	Detection
	Networking features, such as number of requests, data volume, and requests per day, were extracted using TrumanBox. 695 malware samples were executed, and anomalous network behavior (tunneling and backdoors) was detected on these samples. The paper did not provide measures for the TPR or FPR.		
	Trusted system call analysis methodology aimed at the detection of compromised virtual machines using sequential mining [48]	Nissim, 2018	Detection
	Memory dumps were taken from virtual machines running benign or malicious software. System calls were reconstructed from their traces in the stack using WinDbg. 10 malware samples (RATs and ransomware) and six benign software samples were used, with random forest performing the best (98% TPR and 0% FPR).		
Side-Channel	**Machine learning in side-channel analysis: a first study** [97]	Hospodar, 2011	Cryptanalysis
	Attacking cryptographic devices with side-channel analysis is based on the statistic behavior of these devices. This paper demonstrated the use of machine learning to quickly and reliably extract the cryptographic key. The power consumption traces were used as features, and the LS-SVM linear classifier achieved a 75% success rate.		
	On the feasibility of online malware detection with performance counters [15]	Demme, 2013	Classification
	Testing their implementation on Android and Linux, the authors used 210 benign software and 503 malware files (including two rootkits) and tested several algorithms, obtaining a detection rate of 83% and an FPR of 10% with a decision tree algorithm.		
	EDDIE: EM-based detection of deviations in program execution [98]	Nazari, 2017	Detection
	By monitoring the electromagnetic emissions of embedded and IoT devices, the authors were able to detect statistical anomalies in EM emissions caused by injecting code into running applications.		

12 LEVERAGING DYNAMIC ANALYSIS USING MACHINE-LEARNING METHODS

The development of machine-learning methods has contributed greatly to the tasks of malware detection and classification. Such research is based on leveraging information extracted during analysis to gain useful insights into the malware's behavior.

Table 2 summarizes notable papers that implemented machine learning to enhance malware analysis. For each paper, the lead author, year, and task given to the algorithm are listed.

Note that in this section, we present relevant papers that use machine-learning methods in a variety of platforms (Windows, Linux, Android, and the IoT), since our goal was to show the effectiveness of machine-learning methods in all types of dynamic malware analysis techniques. The left column represents the dynamic analysis technique used to extract the features.

13 DISCUSSION AND CONCLUSIONS

As our survey shows, dynamic malware analysis is an ever-evolving domain in which significant progress was made in recent years, as reflected in enhanced performance, higher detection rates, and improved resilience against evasion techniques.

We describe the advancements made in analysis techniques during this time. Early research centered on function call analysis, execution control, and flow tracking. In the past decade, the focus shifted to analyzing the traces left by malware during its execution, and most recently, progress was made in the subdomain of side-channel analysis, which shows promise given the increasing popularity of IoT devices (which cannot be analyzed using traditional techniques), the development of machine-learning algorithms, and the increase in computational power provided by today's CPUs. In terms of secured and trusted analysis, *volatile memory forensics* is more trusted than the *bare metal, VM, hypervisor*, and *emulator* layouts. Although *side-channel analysis* is also highly trusted, it only provides a statistical view of the system, and thus it is primarily useful for analyzing firmware and small devices (such as IoT devices).

Some of the papers surveyed were not well evaluated, as the researchers did not include benign files in their datasets. This means that the method's false-positive rate is unknown. For the benefit of the entire scientific community, we encourage researchers to include their evaluation data (including the specific algorithms and data used) in their published papers so that others can thoroughly assess their work. In addition, test sets should include both malicious and benign files in a ratio aligned with reality. Only three papers included a comparison to previous work, which would allow the reader to assess the superiority of the proposed tool over existing solutions. Without including such a comparison, the research cannot receive its appropriate recognition. In addition, the source-code has not been made available for most of the surveyed tools. Academic tools should be open-sourced so the entire community can make use of them. Furthermore, each tool should include plugin support, so other developers and researchers can leverage existing code to improve the analysis process.

In this survey, we showed that machine learning can be effective at leveraging the output of analysis tools, since increasing numbers of algorithms are applied for malware detection and classification tasks.

14 FUTURE WORK

Deep neural networks have the ability to analyze vast amounts of data. In light of that, we suggest leveraging this ability and creating a large and representative neural network based on system calls. A well-trained network based on a variety of malware samples will enable the detection of malware in different types of files (executables, documents, volatile memory dumps) using the same basic building blocks (i.e., *system calls*).

We presume that this will also address the problem of detecting scarce malware. We hope to show that by training new models on a large variety of malware families, it is possible to detect unknown or scarce malware.

Alternatively, we plan to show that transfer learning can be helpful when dealing with slow or computationally expensive dynamic analysis techniques. We believe that learning on system

calls extracted using one analysis technique (such as function call analysis) will enhance detection using other techniques (such as volatile memory forensics). The latter techniques will only be used for the detection phase, thus saving significant effort during training.

Another avenue of future research stems from the fact that dynamic analysis produces a time sequence output of observed behavior. Such output has been used with sequence mining algorithms to find frequent relations and subsequences between the malware behavior (as observed during analysis) and the malicious activity it correlates to. While sequence mining has limited abilities in finding complex and informative internal relations between the measured features, time-oriented analysis (temporal analysis) is not limited in that way.

In recent years, temporal analysis was found to be very effective in many domains, especially in the biomedical informatics domain. Such an approach can shed new light on malware behavior by discovering frequent time-oriented patterns that differentiate between malicious and benign behavior. Since every piece of data collected during dynamic malware analysis has a timestamp, we believe temporal analysis has great potential in creating advanced machine-learning solutions for the malware detection and categorization tasks.

APPENDIX

A SOFTWARE ANALYSIS PLATFORMS

- *Pin* [99] is a ***binary instrumentation*** (see Section 9.2) platform developed by Intel for code analysis. Instead of simply executing the given code, *Pin* uses a just-in-time (JIT) compiler to read the binary code of the software and transform it into new code. This new code performs the same operations as the original but can also be instrumented to do other tasks (such as ***data tainting***). *Pin* provides a rich API to create plugins (called *PinTools*), which can be shared as open-source extensions. *Pin* does not contain any malware analysis capabilities, but several *PinTools* have been developed to do so, and they can be found on *GitHub*.
- Open-source platforms such as *Xen* [100] (hypervisor) and *QEMU* [101] (emulator) have been used by many analysis tools, since they provide cheap and reliable infrastructure for academic research.
- *TEMU* [102] is a dynamic analysis platform that provides an environment for implementing several analysis techniques. Originally a component within the *BitBlaze project* [103], it has since become an independent project. Its core system is based on system emulation using *QEMU*, an execution engine, and a semantic extractor (to *bridge the semantic gap*).
- *WiLDCAT* [104] is a malware analysis framework built from several components: ***VAMPiRE, Cobra, SPiKE*** (covered in this survey), and ***Sakthi***, which is a cross-operating system ***binary instrumentation*** framework.
- The *Volatility* framework is an open-source ***volatile memory analysis*** tool written in Python. It can process memory dumps taken from Windows XP and up, and most versions of Linux and Mac. *Bridging the semantic gap* is made easy using *Volatility*.

REFERENCES

[1] T. Bell. 1999. The concept of dynamic analysis. *ACM SIGSOFT Softw. Eng. Notes.* 24, 6 (1999), 216–234.
[2] D. Uppal, V. Mehra, and V. Verma. 2014. Basic survey on malware analysis, tools and techniques. *Int. J. Comput. Sci. Appl.* 4, 1 (2014), 103–112.
[3] N. Idika and A. P. Mathur. 2007. A survey of malware detection techniques. Purdue University, 48, 2007-2.
[4] E. Gandotra, D. Bansal, and S. Sofat. 2014. Malware analysis and classification: A survey. *J. Inf. Secur.* 5, 2 (2014), 56–64.
[5] K. Mathur and S. Hiranwal. 2013. A survey on techniques in detection and analyzing malware executables. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.* 3, 4 (2013), 422–428.

[6] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. 2012. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.* 44, 2 (2012), 1–42.
[7] M. Damshenas, A. Dehghantanha, and R. Mahmoud. 2013. A survey on malware propagation, analysis and detection. *Int. J. Cyber-Security Digit. Forens.* 2, 4 (2013), 10–29.
[8] J. Landage and M. Wankhade. 2013. Malware and malware detection techniques: A survey. *Int. J. Eng. Res.* 2, 12 (2013), 61–68.
[9] Y. Ye, T. Li, D. Adjeroh, and S. S. Iyengar. 2017. A survey on malware detection using data-mining techniques. *ACM Comput. Surv.* 50, 3 (2017), 1–40.
[10] L. Yuan, W. Xing, H. Chen, and B. Zang. 2011. Security breaches as PMU deviation: Detecting and identifying security attacks using performance counters. In *Proceedings of the Asia-Pacific Systems Workshop (APSYS'11)*. 6:1–6:5.
[11] D. L. Oppenheimer and M. R. Martonosi. 1997. Performance signatures: A mechanism for intrusion detection. In *Proceedings of the 1997 IEEE Information Survivability Workshop*. 2–5.
[12] S. Vogl and C. Eckert. 2012. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the European Workshop on Systems Security*.
[13] S. S. Clark et al. 2013. Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *Proceedings of the USENIX Workshop on Health Information Technology*. 221–236,
[14] X. Wang and R. Karri. 2013. Numchecker: Detecting kernel control-flow modifying rootkits by using hardware performance counters. In *Proceedings of the 50th Annual Des. Autom. Conference (DAC'13)*. 1–7.
[15] J. Demme et al. 2013. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Comput. Archit. News*. 41, 3 (2013), 559.
[16] A. Tang, S. Sethumadhavan, and S. J. Stolfo. 2014. Unsupervised anomaly-based malware detection using hardware features. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, Cham, 109–129.
[17] S. Vomel and F. C. Freiling. 2011. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digit. Investig.* 8, 1 (2011), 3–22.
[18] M. Graziano, D. Canali, L. Bilge, A. Lanzi, and D. Balzarotti. 2015. Needles in a haystack: Mining information from public dynamic analysis sandboxes for malware intelligence. In *Proceedings of the 24th USENIX Security Symposium (USENIXSecur'15)*, 1057–1072.
[19] K. Rieck, P. Trinius, C. Willems, and T. Holz. 2011. Automatic analysis of malware behavior using machine learning. *J. Comput. Secur.* 19, 4 (2011), 639–668
[20] I. Firdausi, C. lim, A. Erwin, and A. S. Nugroho. 2010. Analysis of machine learning techniques used in behavior-based malware detection. In *Proceedings of the 2nd International Conference Advances on Computing Control, and Telecommunications Technology*. 201–203.
[21] T.-Y. Wang, C.-H. Wu, and C.-C. Hsieh. 2006. A surveillance spyware detection system based on data-mining methods. In *2006 IEEE International Conference on Evolutionary Computation*. IEEE, 3236–3241.
[22] L. Ablon, A. Bogart, and Zero Days. 2017. Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits. Rand Corporation.
[23] P. Regulation. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council. REGULATION (EU), 679, 2016.
[24] E. Toch et al. 2018. The privacy implications of cyber security systems: A technological survey. *ACM Comput. Surv. Artic.* 51, 36 (2018).
[25] M. D. Schroeder and J. H. Saltzer. 1972. A hardware architecture for implementing protection rings. *Commun. ACM* 15, 3 (1972), 157–170.
[26] S. Aboughadareh, C. Csallner, and M. Azarmi. 2014. Mixed-mode malware and its analysis. In *Proceedings of the 4th Progr. Prot. Reverse Eng. Workshop– (PPREW'14)*. 1–12.
[27] B. G. Hoglund and J. Butler. 2005. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley.
[28] J. Schiffman and D. Kaplan. 2014. The SMM rootkit revisited: Fun with USB. In *Proceedings of the 9th International Conference on Availability, Reliab. Secur. (ARES'14)*. 279–286.
[29] A. Sergeev, V. Minchenkov, and V. Bashun. 2015. Malicious hypervisor and hidden virtualization of operation systems. In *Proceedings of the 9th International Conference on Appl. Inf. Commun. Technol. (AICT'15)*. 178–182.
[30] H. Fritsch. 2008. Analysis and detection of virtualization-based rootkits. Technische Universitat, Munchen.
[31] S. T. King, P. M. Chen, Y. M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch. 2006. SubVirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Secur. Priv.* 2006, 314–327.
[32] A. Tereshkin and R. Wojtczuk. 2009. Introducing Ring -3 Rootkits. In *Proceedings of the Black Hat USA Conference (BHUSA'09)*.
[33] M. Gorobets, O. Bazhaniuk, A. Matrosov, A. Furtak, and Y. Bulygin. 2015. Attacking hypervisors via firmware and hardware. Black Hat USA.
[34] N. Nissim, R. Yahalom, and Y. Elovici. 2017. USB-based attacks. *Comput. Secur.* 70 (2017), 675–688.

[35] P. Stewin and I. Bystrov. 2013. Understanding DMA malware. *Lect. Notes Comput. Sci.* 7591 (2013) 21–41.
[36] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. 2009. A fistful of red pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Conference on Offensive Technology*. 2.
[37] J. Rutkowska. 2004. Redpill: Detect VMM using (almost) One CPU Instruction. http://invisiblethings.org/papers/redpill.html.
[38] P. Ferrie. 2007. Attacks on more virtual machine emulators. *Symantec Technol. Exch.* 55.
[39] M. Carpenter, T. Liston, and E. Skoudis. 2007. Hiding virtualization from attackers and malware. *IEEE Secur. Priv.* 5, 3 (2007), 62–65.
[40] I. You and K. Yim. 2010. Malware obfuscation techniques: A brief survey. In *Proceedings of the International Conference on Broadband, Wirel. Comput. Commun. Appl. (BWCCA'10)*. 297–300.
[41] U. Bayer, E. Kirda, and C. Kruegel. 2010. Improving the efficiency of dynamic malware analysis. In *Proceedings of the ACM Symp. Appl. Comput. (SAC'10)*. 1871.
[42] P. Deshpande. 2013. Metamorphic Detection Using Function Call Graph Analysis. Master's Projects. 336. DOI: https://doi.org/10.31979/etd.t9xm-ahsc
[43] J. A P. Marpaung, M. Sain, and H.-J. Lee. 2012. Survey on malware evasion techniques: State of the art and challenges. In *Proceedings of the 14th International Conference on Advances in Communications Technology (ICACT'12)*. 744–749.
[44] B. S. Rivera and R. U. 2015. Inocencio. *Doing More with Less: A Study of Fileless Infection Attacks.*
[45] G. Schwenk and K. Rieck. 2011. Adaptive detection of covert communication in HTTP requests. In *Proceedings of the 7th European Conference on Comput. Netw. Defense (EC2ND'11)*. 25–32.
[46] X. Li, Y. Wen, M. H. Huang, and Q. Liu. 2011. An overview of bootkit attacking approaches. In *Proceedings of the 7th International Conference on Mob. Ad-hoc Sens. Networks (MSN'11)*. 428–431.
[47] Y. Zhu, S. L. Liu, H. Lu, and W. Tang. 2013. Research on the detection technique of Bootkit. In *Proceedings of the International Conference on Graph. Image Process (ICGIP'13)*. 1–7.
[48] N. Nissim, Y. Lapidot, A. Cohen, and Y. Elovici. 2018. Trusted system-calls analysis methodology aimed at detection of compromised virtual machines using sequential mining. *Knowl.-Based Syst.* 153, (2018), 147–175.
[49] A. Cohen and N. Nissim. 2018. Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory. *Expert Syst. Appl.* 102 (2018), 158–178.
[50] D. Kirat, G. Vigna, and C. Kruegel. 2011. BareBox: Efficient malware analysis on bare metal. In *Proceedings of the 27th Annual Comput. Secur. Appl. Conference*. 403–412.
[51] D. D. Chen, M. Egele, M. Woo, and D. Brumley. 2016. Towards fully automated dynamic analysis for embedded firmware. In *Proceedings of the Netw. Distrib. Syst. Secur. Symposium*. 21–24.
[52] J. Rutkowska. 2007. Beyond the CPU: Defeating hardware-based RAM acquisition. In *Proceedings of the Black Hat DC*. 1–49.
[53] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi. 2012. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proceedings of the ACM Comput. Secur. Appl. Conference*. 79.
[54] J. Stuttgen and M. Cohen. 2013. Anti-forensic resilient memory acquisition. *Digit. Investig.* 10, Suppl. S105–S115.
[55] S. Agarwal. FRAME? Framework for real time analysis of malware. In *Proceedings of the 8th International Conference on Cloud Comput. Data Sci. Eng.* 14–15.
[56] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. 2006. Dynamic analysis of malicious code. *J. Comput. Virol.* 2, 1 (2006), 67–77.
[57] T. Mandl, U. Bayer, and F. Nentwich. 2009. ANalyzing Unknown BInarieS the automatic Way. In *Virus Bulletin Conference*, Vol. 1. 2 pages.
[58] G. Willems, T. Holz, and F. Freiling. 2007. Toward automated dynamic malware analysis using CWSandbox. *IEEE Secur. Priv.* 5, 2 (2007), 32–39.
[59] C. Seifert, R. Steenson, I. Welch, P. Komisarczuk, and B. Endicott-Popovsky. 2007. Capture - A behavioral analysis tool for applications and documents. *Digit. Investig.* 4, Suppl. 23–30.
[60] A. Vasudevan. 2008. MalTRAK: Tracking and eliminating unknown malware. In *Proceedings of the Annual Computer Secur. Appl. Conference (ACSAC'08)*. 311–321.
[61] M. Neugschwandtner, C. Platzer, P. M. Comparetti, and U. Bayer. 2010. dAnubis—Dynamic device driver analysis based on virtual machine introspection. In *Proceedings of the Detect. Intrusions Malware, Vulnerability Assess.* 41–60.
[62] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the IEEE Symposium on Secur. Priv.* 317–331.
[63] A. Moser, C. Kruegel, and E. Kirda. 2007. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium Secur. Priv.* 231–245.
[64] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. 2010. Efficient detection of split personalities in malware. In *Proceedings of the Conference on Netw. Distrib. Syst. Secur.*

[65] D. Bruening. 2004. Efficient, transparent, and comprehensive runtime code manipulation. *Electr. Eng.* 306.
[66] A. Vasudevan and R. Yerraballi. 2005. Stealth breakpoints. In *Proceedings of the Annual Computer Secur. Appl. Conference (ACSAC'05)*. 381–390.
[67] A. Vasudevan and R. Yerraballi. 2006. SPiKE: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the Conference on Res. Pract. Inf. Technol. Ser.* 48, 311–320.
[68] A. Vasudevan and R. Yerraballi. 2006. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proceedings of the IEEE Symposium on Secur. Priv.* (2006), 264–278.
[69] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. 2007. Automatically identifying trigger-based behavior in malware. In *Proceedings of the Conference on Botnet Detect.* 65–88.
[70] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. 2008. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'08)*. 11.
[71] M. Polino et al. 2017. Measuring and defeating anti-instrumentation-equipped malware. In *Proceedings of the International Conference on Detect. Intrusions Malware, Vulnerability Assess.* 73–96.
[72] S. Cesare and Y. Xiang. 2010. Classification of malware using structured control flow. In *Proceedings of the Conference on Res. Pract. Inf. Technol. Ser.* 107, AusPDC, 61–70, 2010.
[73] M. Costa et al. 2005. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the Symposium on Operating Systems Principles (SOSP'05)*.
[74] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference Comput. Commun. Secur. (CCS'07)*, 116–127.
[75] J. Clause, W. Li, and A. Orso. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Softw. Test. Anal.* 196–206, 2007.
[76] M. Egele, C. Kruegel, E. Kirda, and D. Song. 2007. Dynamic spyware analysis. In *Usenix Conference*. 233–246.
[77] I. Korkin and I. Nesterov. 2014. Applying memory forensics to rootkit detection. In *Proceedings of the Conference on Digit. Forensics, Secur. Law*, 115–142.
[78] J. Stüttgen, S. Vomel, and M. Denzel. 2015. Acquisition and analysis of compromised firmware using memory forensics. *Digit. Investig.* 12, S1, S50–S60.
[79] Z. Liang, H. Yin, and D. Song. 2008. HookFinder: Identifying and understanding malware hooking behaviors. *Dep. Electr. Comput. Eng.* 41 (2008).
[80] J. Rhee, R. Riley, D. Xu, and X. Jiang. 2010. Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory. *Lect. Notes Comput. Sci. (Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 6307 (2010), 178–197.
[81] C. Gorecki, F. C. Freiling, K. Marc, and T. Holz. 2011. Truman box: Improving dynamic malware analysis by emulating the internet. In *Symposium on Self-Stabilizing Systems*. Springer, Berlin, Heidelberg, 208–222.
[82] M. Yu, Z. Qi, Q. Lin, X. Zhong, B. Li, and H. Guan. 2012. Vis: Virtualization enhanced live forensics acquisition for native system. *Digit. Investig.* 9, 1 (2012), 22–33.
[83] M. Graziano, A. Lanzi, and D. Balzarotti. 2013. Hypervisor memory forensics. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, Berlin, Heidelberg, 21–40.
[84] A. Mohaisen, O. Alrawi, and M. Mohaisen. 2015. AMAL: High-fidelity, behavior-based automated malware analysis and classification. *Comput. Secur.* 52 251–266.
[85] Y. Cheng, X. Fu, X. Du, B. Luo, and M. Guizani. 2017. A lightweight live memory forensic approach based on hardware virtualization. *Info. Sci.* 379, 23–41.
[86] C. H. Lin, C. W. Tien, C. W. Chen, C. W. Tien, and H. K. Pao. 2016. Efficient spear-phishing threat detection using hypervisor monitor. In *Proceedings of the International Carnahan Conference Secur. Technol.* 299–303.
[87] C. H. Lin, H. K. Pao, and J. W. Liao. 2018. Efficient dynamic malware analysis using virtual time control mechanics. *Comput. Secur.* 73 359—373.
[88] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose. SoK? The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In *Proceedings of 40th IEEE Symposium on Security and Privacy (S&P'19)*.
[89] E. Buchanan, R. Roemer, S. Savage, and H. Shacham. 2008. Return-oriented programming: Exploitation without Code Injection. Black Hat, 8.
[90] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi. 2018. Hardware performance counters can detect malware: myth or fact? In *Proceedings of the Conference on Asia Conference on Computer and Communications Security (ASIACCS'18)*. 457–468.
[91] C. Smith and S. S. Consultant. 2008. Creating code obfuscation virtual machines. Tutorial in RECON08.
[92] M. M. G. Kang, P. Poosankam, and H. Yin. 2007. Renovo: A hidden code extractor for packed executables. In *Proceedings of the ACM Workshop on Recurr. Malcode* 46–53.

[93] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. 2006. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the Annual Computer Secur. Appl. Conference (ACSAC'06)*. 289–298.
[94] R.O.C.U. 2005. United. Generic Unpacking—How to Handle Modified or Unknown PE Compression Engines. Retrieved from https://www.virusbulletin.com/conference/vb2005/abstracts/generic-unpacking-how-handle-modified-or-unknown-pe-compression-engines/.
[95] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. 2015. Malware classification with recurrent networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'15)*. 1916–1920.
[96] S. Rasthofer, S. Arzt, and E. Bodden. 2014. A Machine-learning approach for classifying and categorizing android sources and sinks. In *Proceedings of the Network and Distributed System Security Symposium*. 23–26.
[97] G. Hospodar, B. Gierlichs, E. De Mulder, I. Verbauwhede, and J. Vandewalle. 2011. Machine learning in side-channel analysis: A first study. *J. Cryptogr. Eng.* 1, 4 (2011), 293–302.
[98] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. 2017. EDDIE: EM-based detection of deviations in program execution. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*. 333–346.
[99] C.-K. Luk et al. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Program. Lang. Des. Implement. (PLDI'05)*. 190.
[100] P. Barham et al. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operations Systems Principles (SOSP'03)*. 164.
[101] F. Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*. 41–46.
[102] H. Yin and D. Song. 2012. Automatic Malware Analysis: An Emulator Based Approach. Springer Science & Business Media.
[103] D. Song et al. 2008. BitBlaze: A new approach to computer security via binary analysis. *Lect. Notes Comput. Sci.* 5352 (2008), 1–25.
[104] A. Vasudenvan. 2007. WiLDCAT: An integrated stealth environment for dynamic malware analysis. PhD Dissertations. http://hdl.handle.net/10106/233.

Received November 2018; revised April 2019; accepted May 2019