6

A Survey on Automated Dynamic Malware-Analysis Techniques and Tools

MANUEL EGELE, Vienna University of Technology
THEODOOR SCHOLTE, SAP Research, Sophia Antipolis
ENGIN KIRDA, Institute Eurecom, Sophia Antipolis
CHRISTOPHER KRUEGEL, University of California, Santa Barbara

Anti-virus vendors are confronted with a multitude of potentially malicious samples today. Receiving thousands of new samples every day is not uncommon. The signatures that detect confirmed malicious threats are mainly still created manually, so it is important to discriminate between samples that pose a new unknown threat and those that are mere variants of known malware.

This survey article provides an overview of techniques based on dynamic analysis that are used to analyze potentially malicious samples. It also covers analysis programs that employ these techniques to assist human analysts in assessing, in a timely and appropriate manner, whether a given sample deserves closer manual inspection due to its unknown malicious behavior.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Dynamic analysis, malware

ACM Reference Format:
Egele, M., Scholte, T., Kirda, E., and Kruegel, C. 2012. A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. 44, 2, Article 6 (February 2012), 42 pages.
DOI = 10.1145/2089125.2089126 http://doi.acm.org/10.1145/2089125.2089126

1. INTRODUCTION

The Internet has become an essential part of daily life as more and more people use services that are offered on the Internet. The Internet has evolved from a basic communication network to an interconnected set of information sources enabling, new forms of (social) interactions and marketplaces for the sale of products and services among other things. Online banking or advertising are examples of the commercial aspects of the Internet. Just as in the physical world, there are people on the Internet with malevolent intents, who strive to enrich themselves by taking advantage of legitimate users whenever money is involved. Malware (i.e., software of malicious intent) helps these people accomplish their goals.

To protect legitimate users from these threats, security vendors offer tools that aim to identify malicious software components. Typically, these tools apply some sort of signature matching process to identify known threats. This technique requires the

This work has been supported by the European Commission through project FP7-ICT-216026-WOMBAT, by FIT-IT through the SECoverer project, and by Secure Business Austria.
Corresponding author's address: M. Egele, Automation Systems Group (E183-1), Vienna University of Technology, Treitlstr. 1, 1040 Vienna, Austria; email: manuel@seclab.tuwien.ac.at.

DOI 10.1145/2089125.2089126 http://doi.acm.org/10.1145/2089125.2089126

vendor to provide a database of signatures which are then compared against potential threats. Once the security vendor obtains a sample of a new potential threat to study, the first step is for a human analyst to determine whether this (so far unknown) sample poses a threat to users by analyzing the sample. If the sample poses a threat, the analyst attempts to find a pattern that allows her to identify this sample (i.e., the signature). This pattern should be generic enough to also match with variants of the same threat, but not falsely match on legitimate content. The analysis of malware and the successive construction of signatures by human analysts is time-consuming and error-prone. It is also trivial for malware authors to automatically generate a multitude of different malicious samples derived from a single malware instance. It is not extraordinary for an anti-virus vendor to receive thousands of unknown samples per day. Symantec [Fossi et al. 2009] (averaging 4,300 per day) as well as McAfee [Marcus et al. 2009] (averaging 12,300 per day) report receiving over 1.6M new samples in 2008. This substantial quantity requires an automated approach to quickly differentiating between samples that deserve closer (manual) analysis and those that are a variation of an already known threats. This automatic analysis can be performed in two ways. *Dynamic* analysis refers to techniques that execute a sample and verify the actions this sample performs in practice, while *static* analysis performs its task without actually executing the sample.

This article focuses on the techniques that can be applied to analyze potential threats and discriminate samples that are mere variations of already known threats. In addition, it presents the currently available tools and their underlying approaches to performing automated dynamic analysis on potentially malicious software.

2. WHAT IS MALWARE?

Software that "deliberately fulfills the harmful intent of an attacker" is commonly referred to as malicious software or malware [Moser et al. 2007a]. Terms such as "worm," "virus," or "Trojan horse" are used to classify malware samples that exhibit similar malicious behavior. The first instances of malicious software were viruses. The motivation for the creators of such early malware was usually to highlight some security vulnerabilities or simply to show off technical ability. For example, the cleaning of Bagle worm-infected hosts by instances of the Netsky worm could be considered a rivalry between different authors [Tanachaiwiwat and Helmy 2006]. As time passed, the motivations changed. Today there is a flourishing underground economy based on malware [Zhuge et al. 2008]. It is no longer the fun factor that drives the development in these circles, but the prospect of money that can be made.

Consider the following scenario which illustrates the distribution of malware and its effects. A bot is a remotely-controlled piece of malware that has infected an Internet-connected computer system. This bot allows an external entity, the so-called bot master, to remotely control this system. The pool of machines that are under control of the bot master is called a botnet. The bot master might rent this botnet to a spammer who misuses the bots to send spam emails containing links to a manipulated webpage. This page, in turn, might surreptitiously install a spyware component on a visitors system which collects personal information, such as credit card numbers and online banking credentials. This information is sent back to the attacker who is now able to misuse the stolen information by purchasing goods online. All involved criminals make money at the expense of the infected user, or her bank, respectively. With the rise of the Internet and the number of attached hosts, it is now possible for a sophisticated attacker to infect thousands of hosts within hours after releasing the malware into the wild [Moore et al. 2002]. Recently, a study by Stone-Gross et al. [2009] revealed that the Torpig botnet consists of more than 180,000 infected computer systems.

The risk described above motivates the need to create tools that support the detection and mitigation of malicious software. Nowadays, the weapon of choice in combat against malicious software is signature-based anti-virus scanners that match a pregenerated set of signatures against the files of a user. These signatures are created in a way so that they only match malicious software. This approach has at least two major drawbacks. First, the signatures are commonly created by human analysts. This often is a tedious and error-prone task. Second, the usage of signatures inherently prevents the detection of unknown threats for which no signatures yet exist. Thus, whenever a new threat is detected, it needs to be analyzed, and signatures need to be created for this threat. After the central signature database has been updated, the new information needs to be deployed to all clients that rely on that database. Because the signatures are created by human analysts, unfortunately, there is room for error. Multiple AV vendors released signature updates that mistakenly identified legitimate executables as malware [FRISK Software International 2003; John Leyden (The Register) 2007], thus rendering the operating system they were designed to protect, inoperative.

Closely related to the second drawback (i.e., not being able to detect unknown threats), is the inability to detect specifically tailored malware. Besides the mass phenomenon of Internet worms and malicious browser plug-ins, one can observe the existence of specifically tailored malware created for targeted attacks [Avira Press Center 2007]. Spyware programs may be sent via email to the executive board of a company with the specific intent to capture sensitive information regarding the company. Because these malware samples usually do not occur in the wild, it is unlikely that an anti-virus vendor receives a sample in time to analyze it and produce signatures. This means that the spyware could be operational in the company for a long time before it is detected and removed, even if anti-virus software is in place.

The inability to detect unknown threats is an inherent problem of signature-based detection techniques. This is overcome by techniques that base their decision of identifying a piece of code as being malicious or not, on the observation of the software's behavior. Although these techniques allow for the detection of previously unknown threats to a certain extent, they commonly suffer from false positives. That is, legitimate samples are falsely classified by the detection system as being malicious due to the detector's inability to distinguish legitimate from malicious behavior under all circumstances.

2.1. Types of Malware

This section gives a brief overview of the different classes of malware programs that have been observed in the wild. The following paragraphs are solely intended to familiarize the reader with the terminology that we will be using in the remainder of this work. Furthermore, these classes are not mutually exclusive. That is, specific malware instances may exhibit the characteristics of multiple classes at the same time. A more detailed discussion of malicious code in general can be found in Skoudis and Zeltser [2003] or Szor [2005].

Worm. Prevalent in networked environments, such as the Internet, Spafford [1989] defines a worm as "a program that can run independently and can propagate a fully working version of itself to other machines." This reproduction is the characteristic behavior of a worm. The Morris Worm [Spafford 1989] is the first publicly known instance of a program that exposes worm-like behavior on the Internet. More recently in July 2001, the Code Red worm infected thousands (359,000) of hosts on the Internet during the first day of its release [Moore et al. 2002]. Today, the Storm worm and others are used to create botnets that are rented out by bot masters to send spam emails or perform distributed denial of service attacks (DDOS) [Kanich et al. 2008], in which

multiple worm-infected computers try to exhaust the system resources or the available network bandwith of a target in a coordinated manner.

Virus. "A virus is a piece of code that adds itself to other programs, including operating systems. It cannot run independently—it requires that its 'host' program be run to activate it." [Spafford 1989]. As with worms, viruses usually propagate themselves by infecting every vulnerable host they can find. By infecting not only local files but also files on a shared file server, viruses can spread to other computers as well.

Trojan Horse. Software that pretends to be useful but performs malicious actions in the background is called a Trojan horse. While a Trojan horse can disguise itself as any legitimate program, frequently, they pretend to be useful screensavers, browser plug-ins, or downloadable games. Once installed, their malicious part might download additional malware, modify system settings, or infect other files on the system.

Spyware. Software that retrieves sensitive information from a victim's system and transfers this information to the attacker is denoted as spyware. Information that might be interesting for the attacker include accounts for computer systems or bank account credentials, a history of visited web pages, and contents of documents and emails.

Bot. A bot is a piece of malware that allows its author (i.e., the bot master) to remotely control the infected system. The set of bots collectively controlled by one bot master is a *botnet*. Bots are commonly instructed to send spam emails or perform spyware activities as just described.

Rootkit. The main characteristic of a rootkit is its ability to hide certain information (i.e., its presence) from a user of a computer system. Rootkit techniques can be applied at different system levels, for example, by instrumenting API calls in user-mode or tampering with operating system structures if implemented as a kernel module or device driver. Manipulating the respective information allows a rootkit to hide processes, files, or network connections on an infected system. Moreover, virtual machine-based rootkits [King et al. 2006; Rutkowska 2006; Zovi 2006] conceal their presence by migrating an infected operating system onto a virtual machine. The hiding techniques of rootkits are not bad per se, but the fact that many malware samples apply rootkit techniques to hide their presence in the system justifies mentioning them here.

2.2. Infection Vectors

This section gives an overview of the infection vectors commonly used by malicious software to infect a victim's system. Brief examples are used to illustrate how these infections work and how malware used them in the past.

2.2.1. Exploiting Vulnerable Services over the Network. Network services running on a server provide shared resources and services to clients in a network. For example, a DNS service provides the capabilities of resolving host names to IP addresses, and a file server provides shared storage on the network. Many commodity off-the-shelf operating systems come with a variety of network services that are already installed and running. Vulnerabilities in such services might allow an attacker to execute code on the machine that is providing the service. Large installation bases of services that share the same vulnerability (e.g., [Microsoft Corporation 2008]) paved the way for automatic exploitation. Thus, such conditions allow malicious software to infect accessible systems automatically. This characteristic makes network service exploitation the preferred method for infection by worms. Moreover, services that provide system access to remote users and authenticate these users with passwords (e.g., ssh, administrative Web interfaces, etc.), are frequently exposed to so-called *dictionary attacks*. Such an attack iteratively tries to log into a system using passwords stored in a dictionary.

2.2.2. Drive-by Downloads. Drive-by downloads usually target a victim's Web browser. By exploiting a vulnerability in the Web browser application, a drive-by download is

able to fetch malicious code from the Web and subsequently execute it on the victim's machine. This usually happens without further interaction with the user. In contrast to exploiting vulnerabilities in network services in which push-based infection schemes are dominant, drive-by downloads follow a pull-based scheme [Provos et al. 2008]. That is, the connections are initiated by the client as it is actively requesting the malicious contents. Therefore, firewalls that protect network services from unauthorized access cannot mitigate the threat of drive-by attacks. Currently, the following two techniques are observed in the wild which might lead to drive-by infections.

—*API misuse.* If a certain API allows for downloading an arbitrary file from the Internet, and another API provides the functionality of executing a random file on the local machine, the combination of these two APIs can lead to a drive-by infection [Microsoft Corporation 2006]. The widespread usage of browser plug-ins usually gives attackers a huge portfolio of APIs that they might use and combine for their nefarious purposes in unintended ways.

—*Exploiting Web browser vulnerabilities.* This attack vector is identical to the case of exploitable network services. Moreover, as described in Daniel and Sotiroy [2008], the availability of client-side scripting languages, such as Javascript or VBScript, provide the attacker with additional means to successfully launch an attack.

Before a drive-by download can take place, a user is first required to visit the malicious site. In order to lure the user into visiting the malicious site, attackers perform social engineering and send spam emails that contain links to these sites or infect existing webpages with the malicious code. For example, the infamous Storm worm makes use of its own botnet resources to send spam emails containing links to such attack pages [Kanich et al. 2008].

To maximize the number of sites that host such drive-by attacks, attackers exploit vulnerabilities in Web applications that allow them to manipulate these websites (e.g., [Dan Goodin (The Register) 2008]). This is an example in which attackers use the infection vector of vulnerable network services to launch drive-by download attacks on clients of that service (e.g., a website). Another technique for attackers to lure users to their websites is by trying to cheat the ranking algorithms Web search engines use, to sort result pages. An attacker may create a page that is specifically instrumented to rank high for common search query terms. If the page is listed on a top position for these query terms, it will result in a large number of visitors [Cacheda and Viña 2001; Jansen and Spink 2005]. Provos et al. [2008] discovered that more than 1.3% of results to Google search queries include at least one page that tries to install malicious software on a visitor's machine. Provos et al. [2007] also analyzed the techniques malware authors apply to lure a user to open a connection to a host that performs drive-by download attacks. The most prevalent forms of such actions are circumventing Web-server security measures, providing user generated content, advertising schemes, and malicious widgets.

2.2.3. Social Engineering. All techniques that lure a user into deliberately executing malicious code on her machine, possibly under false pretenses, are subsumed as social engineering attacks. There are virtually no limits to the creativity of attackers when social engineering is involved. Asking the user to install a provided "codec" to view the movie that is hosted on the current web site, clicking and opening an image that is attached to a spam email, or speculating that the user eventually plugs a "found" USB key into her computer [Stasiukonis 2007] are just a few examples of social engineering.

2.3. Malware Analysis

Today, signatures for anti-virus toolkits are created manually. Prior to writing a signature, an analyst must know if an unknown sample poses a threat to users. Different

malware-analysis techniques allow the analyst to quickly and in detail understand the risk and intention of a given sample. This insight allows the analyst to react to new trends in malware development or refine existing detection techniques to mitigate the threat coming from that sample. The desire of analysts to understand the behavior of a given sample, and the opposing intention of malware authors to disguise their creation's malicious intents, leads to an arms race between those two parties. As analysis tools and techniques become more elaborate, attackers come up with evasion techniques to prevent their malware from being analyzed. Such techniques cover self-modifying or dynamically generated code, as well as approaches that detect the presence of an instrumented analysis environment, thus allowing the malware to conceal or inhibit its malicious behavior. Before we elaborate on possible evasion mechanisms, the following sections present an overview of applicable program-analysis techniques that are used today to analyze malicious code.

The process of analyzing a given program during execution is called *dynamic analysis*, while *static analysis* refers to all techniques that analyze a program by inspecting it. Dynamic analysis is covered in detail in Section 3. For completeness, a brief introduction and known limitations of static analysis approaches follows.

2.3.1. Static Analysis. Analyzing software without executing it is called static analysis. Static analysis techniques can be applied on different representations of a program. If the source code is available, static analysis tools can help find memory corruption flaws [Chen et al. 2004; Chen and Wagner 2002; Feng et al. 2004] and prove the correctness of models for a given system.

Static analysis tools can also be used on the binary representation of a program. When compiling the source code of a program into a binary executable, some information (e.g., the size of data structures or variables) gets lost. This loss of information further complicates the task of analyzing the code.

Static analysis tools can be used to extract useful information about a program. Call graphs give an analyst an overview of what functions might be invoked from where in the code. If static analysis is able to calculate the possible values of parameters [Egele et al. 2006], this knowledge can be used for advanced protection mechanisms.

Problems of Static Malware Analysis Approaches. Generally, the source code of malware samples is not readily available. That reduces the applicable static analysis techniques for malware analysis to those that retrieve the information from the binary representation of the malware. Analyzing binaries brings along intricate challenges. Consider, for example, that most malware attacks host executing instructions in the IA32 instruction set. The disassembly of such programs might result in ambiguous results if the binary employs self-modifying code techniques (e.g., packer programs as discussed in Section 4.6.1). Additionally, malware relying on values that cannot be statically determined (e.g., current system date, indirect jump instructions) exacerbate the application of static analysis techniques. Moreover, Moser et al. [2007b] proposes an approach that relies on opaque constants to thwart static analysis. It can be expected that malware authors know of the limitations of static analysis methods, and thus, will likely create malware instances that employ these techniques to thwart static analysis. Therefore, it is necessary to develop analysis techniques that are resilient to such modifications and are able to reliably analyze malicious software.

3. DYNAMIC MALWARE ANALYSIS TECHNIQUES

Analyzing the actions performed by a program while it is being executed is called dynamic analysis. This section deals with the different approaches and techniques that can be applied to perform such dynamic analysis.

3.1. Function Call Monitoring

Typically, a function consists of code that performs a specific task, such as calculating the factorial value of a number or creating a file. While the use of functions can result in easy code re-usability, and easier maintenance, the property that makes functions interesting for program analysis is that they are commonly used to abstract from implementation details to a semantically richer representation. For example, the particular algorithm which a sort function implements might not be important as long as the result corresponds to the sorted input. When it comes to analyzing code, such abstractions help gain an overview of the behavior of the program. One possibile way to monitor what functions are called by a program is to intercept these calls. The process of intercepting function calls is called *hooking*. The analyzed program is manipulated in a way so that, in addition to the intended function, a *hook function* is invoked. This hook function is responsible for implementing the required analysis functionality, such as recording its invocation to a log file, or analyze input parameters.

Application Programming Interface (API). Functions that form a coherent set of functionality, such as manipulating files or communicating over the network, are often grouped into a application programming interface (API). Operating systems usually provide many APIs that can be used by applications to perform common tasks. These APIs are available on different layers of abstraction. Network access, for example, can be provided by an API that focuses on the content transmitted in TCP packets, or by a lower-level API that allows the application to create and write packets directly to a raw socket. On Windows operating systems, the term Windows API refers to a set of APIs that provide access to different functional categories, such as networking, security, system services, and management.

System Calls. Software executing on computer systems that run commodity off-the-shelf operating systems, is usually divided in two major parts. While general applications, such as word processors, or image manipulation programs are executed in user-mode, the operating system is executed in kernel-mode. Only code that is executed in kernel-mode has direct access to the system state. This separation prevents user-mode processes from interacting with the system and their environment directly. It is, for example, not possible for a user-space process to directly open or create a file. Instead, the operating system provides a special well-defined API—the *system call* interface. Using system calls, a user-mode application can request the operating system to perform a limited set of tasks on its behalf. Thus, to create a file, a user-mode application needs to invoke the specific system call indicating the file's path, name, and access method. Once the system call is invoked, the system is switched into kernel-mode (i.e., privileged operating system code is executed). Upon verification that the calling application has sufficient access rights for the desired action, the operating system carries out the task on behalf of the user-mode applications. In the case of the file-creation example, the result of the system call is a file handle, where all further interaction of the user-mode application regarding this file (e.g., write to the file) is then performed through this handle. Apart from exhaustively using resources (bound by the limits of the operating system), there is usually little a malware sample can do within the bounds of its process. Therefore, malware (just like as every other application) that executes in user-space and needs to communicate with its environment has to invoke the respective system calls. Since system calls are the only possibility for a user-mode process to access or interact with its environment, this interface is especially interesting for dynamic malware analysis. However, there are known malware samples that manage to gain kernel-mode privileges. Such instances do not necessarily make use of the system call interface and might evade this analysis method.

Windows Native API. The Windows Native API [Nebbett 2000] resides between the system call interface and the Windows API. While the Windows API remains stable for any given version of the Windows operating system, the Native API is not restricted in that way and may change with different service pack levels of the same Windows version. The Native API is commonly invoked from higher-level APIs, such as the Windows API, to invoke the system calls and perform any necessary pre- or postprocessing of arguments or results. Legitimate applications commonly communicate with the OS through the Windows API, but malicious code might skip this layer and interact with the Native API directly to thwart monitoring solutions that hook the Windows API only. This comes, of course, with the additional burden for the malware author to design the malware in a way that it covers all different versions of the Native API. As there is no official and complete documentation of the Native API, this requires great knowledge of Windows internals. Similarly, a malware author might decide to skip the Native API and invoke the system calls directly from the malware. While this is possible, it requires even deeper knowledge of an even less-documented interface.

Results of Function Hooking. Hooking API functions allows an analysis tool to monitor the behavior of a program on the abstraction level of the respective function. While semantically rich observations can be made by hooking Windows API functions, a more detailed view of the same behavior can be obtained by monitoring the Native API. The fact that user-space applications need to invoke system calls to interact with their environment implies that this interface deserves special attention. However, this restriction only holds true for malware running in user mode. Malware executing in kernel mode can directly invoke the desired functions without passing through the system call interface.

3.1.1. Implementing Function Hooking. Depending on the availability of the programs' source code, different approaches to hook functions can be applied. If the source code is available, invocations to hook functions can be inserted into the source code at the appropriate places. Alternatively, compiler flags (e.g., finstrument-functions in GCC [Free Software Foundation]) can be used to implement hooking.

Binary rewriting is used if the program under analysis is only available in binary form. To this end, two approaches can perform the necessary analysis. (1) Rewriting the monitored function in a way such that, prior to executing its original code, the function invokes the hook. (2) Finding and modifying all call sites (i.e., the `call` instructions) that, upon execution, invoke the monitored function to call the hook instead.

In both approaches, the hook function obtains full access to the original arguments on the stack and can perform the desired analysis steps. In addition, if the function is invoked through a function pointer (e.g., functions in shared libraries), this value can be changed to point to the hook function instead. On Windows operating systems the Detours library is readily available to facilitate function call hooking.

The idea behind Detours [Hunt and Brubacher 1999] is to apply target function rewriting to implement function call hooking. This is accomplished by redirecting control flow from the function to hook to the analysis function, which in turn might call the original function. The diversion of the control flow is implemented by overwriting the initial instructions of the original function with an unconditional jump to the analysis code. The overwritten instructions are backed up and copied to a trampoline function. This trampoline consists of the backed-up instructions, and an unconditional jump into the original function after the overwritten instructions. Once the monitored application calls the hooked function, the new instructions divert the control flow to the analysis code. This code can perform any preprocessing (e.g., sanitizing of arguments) and has full control over the control flow. The analysis code can then instantly return to the caller or invoke the original function by calling the trampoline. Since the original

function is called by the analysis code, this code is in control after the function returns and can then perform any postprocessing necessary.

Detours offers two alternative approaches to applying the necessary modifications to a program. (1) It modifies binary files before they are loaded and executed, or (2) it manipulates the in-memory image of an already-loaded binary. The first technique requires adding additional sections to the binary file while it is on disk. This requires modification of the disk structure of the file to make space for the additional code. Modifying an already-running binary is accomplished through DLL injection. First, all payload data (i.e., analysis functions) is compiled into a DLL. Then, a new thread is created in the running binary that loads this DLL. Upon initialization of the DLL, Detours manipulates the binary as just described (e.g., create trampolines, overwrite initial instructions of the target function).

Debugging techniques can also be used to hook into invocations of certain functions. Breakpoints can be inserted at either the call site or the function to monitor. When a breakpoint triggers, control is given to the debugger that has full access to the memory contents and CPU state of the debugged process. Thus, an instrumented debugger can be used to perform the intended analysis.

If available, the hooking infrastructure provided by the operating system can be used to monitor system actions. The Windows operating system, for example, provides mechanisms to specify messages and corresponding hook functions. Whenever an application receives the specified message (e.g., a key stroke occurred on the keyboard, or a mouse button has been pressed), the hook function is executed.

Replacing dynamic shared libraries can serve as another means to perform function call monitoring. For analysis purposes, the original libraries might be renamed and replaced by stub libraries containing the hook functions. These stubs can either simulate the original behavior or call the original function in the renamed library. This method is able to fully capture the interactions of a program with a given API.

3.1.2. Postprocessing Function Call Traces. Monitoring function calls results in a function call trace. This trace consists of the sequence of functions that were invoked by the program under analysis, along with the passed arguments. Different approaches exist that take these function call traces as input and abstract to semantically richer representations of the malware behavior. Christodorescu et al. [2007] convert function call traces to a graph representation. This representation allows them to compare the behavior of malicious software against behavior exhibited by legitimate software. The difference between these graphs represents the malicious core of the malware. Such techniques enable the analyst to detect malicious instances of the same malware family even for previously unknown samples. Xu et al. [2004] compare function call traces of known malicious binaries to those of unknown samples in order to detect polymorphic variants of the same threat. They apply sequence alignment techniques to calculate similarities between function call traces.

3.2. Function Parameter Analysis

While function parameter analysis in static analysis tries to infer the set of possible parameter values or their types in a static manner, dynamic function parameter analysis focuses on the actual values that are passed when a function is invoked. The tracking of parameters and function return values enables the correlation of individual function calls that operate on the same object. For example, if the return value (a file-handle) of a `CreateFile` system call is used in a subsequent `WriteFile` call, such a correlation is obviously given. Grouping function calls into logically coherent sets provides detailed insight into the program's behavior from a different, object-centric, point-of-view.

```
//tainted x
a = x;        mov eax, x
a = a + x;    add eax, x
```

Fig. 1. Direct data dependencies examples.

```
//tainted pointer x   e.g., jump table
a = x[10];            mov eax, [x + 10]
//tainted value y     e.g., translation table
b = c[y];             mov eax, [c + y]
```

Fig. 2. Address dependencies examples.

3.3. Information Flow Tracking

An orthogonal approach to the monitoring of function calls during the execution of a program, is the analysis on how the program processes data. The goal of information flow tracking is to shed light on the propagation of "interesting" data throughout the system while a program manipulating this data is executed. In general, the data that should be monitored is specifically marked (tainted) with a corresponding label. Whenever the data is processed by the application, its taint-label is propagated. Assignment statements, for example, usually propagate the taint-label of the source operand to the target. Besides the obvious cases, policies have to be implemented that describe how taint-labels are propagated in more difficult scenarios. Such scenarios include the usage of a tainted pointer as the base address when indexing to an array or conditional expressions that are evaluated on tainted values.

Taint Sources and Sinks. Two vital concepts in information flow tracking systems are *taint sources* and *taint sinks*. A taint source is responsible for introducing new taint labels into the system (i.e., tainting the data that is deemed interesting by the analysis). In contrast to a taint source, a taint sink is a component of the system that reacts in a specific way when stimulated with tainted input (e.g., emit a warning when tainted data is transferred over the network).

The following examples should give a brief overview of the aspects that need to be taken into account when developing information flow tracking systems. The code snippets use the C syntax, and a corresponding Intel asm notation, where appropriate.

Direct data dependencies. Taint labels are simply propagated for direct assignments or arithmetic operations that are dependent on a tainted value.

Policies need to define what happens if two taint labels are involved in the same operation. Recall the above arithmetic operation (`a = a + x`) in Figure 1. We assume that both operands are tainted with distinct taint labels. The propagation policy has to define what should happen in such a case. Possible scenarios include choosing one label over the other, or creating a new label representing a combination of the two original labels.

Address dependencies. A system that implements address tainting also propagates taint labels for memory-read or -write operations whose target addresses are derived from tainted operands. Translating a value from one alphabet to another, for example, can be effectively accomplished by the use of translation tables. Such a table is used in Windows environments to translate ASCII characters to their Unicode counterparts. When tracking address dependencies, the target of an operation gets tainted if a tainted value is used as a pointer to the base of an array, or as an index into an array. See Figure 2 for an example.

Control flow dependencies. Direct data dependencies and addressing dependencies are similar in that the decision whether the taint label is propagated to the target

```
//tainted x
if (x == 0) {
  v = 0;
} if (x == 1) {
  v = 1;
} ...
```

Fig. 3. Control flow dependencies example.

```
//tainted x
a = 0; b = 0;
if (x == 1) then a = 1 else b = 1;
if (a == 0) then v = 0;
if (b == 0) then v = 1;
```

Fig. 4. Implicit information flow example.

of an instruction can be reached in the scope of a single asm instruction. Correctly tracking information that is propagated based on control flow decisions, is intricate. The following snippet copies the value of `x` to `v` (see Figure 3).

A system equipped with direct data and address-dependency tracking would miss such an information flow since no assignment (direct or indirect) is done via the tainted variable `x`. To cover these cases as well, a possible (conservative) solution is to execute the following two-step algorithm: Whenever a tainted value is involved in a control flow decision, (1) calculate the reconvergence point (i.e., the first instruction that is executed regardless of which branch is executed) of the branch instruction. (2) Until the reconvergence point is reached, every target of an assignment is tainted regardless of the taint label of the source operands. This is implemented, for example, in Egele et al. [2007] and Nair et al. [2008].

Implicit information flow. The following example exploits the semantic relationship between the variables `a,b`, and `x` in a way to copy the value of `x` to `v`, thereby evading the taint-tracking mechanisms so far described.

The code in Figure 4 reflects the implicit information that after the execution of the `if` statement (line 2), *exactly one* of the variables `a` and `b` is set to 1. For example, if `x` holds the value 1, line 2 would set `a` to 1. Subsequently, the `if` statement in line 3 would not execute the `then` branch (since $a \neq 0$). Finally, `v` is assigned the value 1 in line 4. The program behaves similarly when `x` equals zero. Although this sample only leaks one bit of information, a malicious software can execute this code repeatedly to hide arbitrary amounts of data from information flow tracking systems that track data, address, and control flow dependencies.

In order to successfully track the information flow in such cases as the example, the values of `a` and `b` need to be tainted in line 2. This would require that a similar approach for controlling flow dependencies is followed, in which additional assignments in not executed branches need to be tainted. Identifying these assignments can be accomplished either by analyzing the snippet statically (e.g., [Backes et al. 2009; Nair et al. 2008]), or dynamically by forcing execution down all available branches (e.g., [Moser et al. 2007a]).

Implementation of information flow tracking systems. Depending on the application analyzed, information flow tracking can be implemented on different levels.

For interpreted languages (e.g., JavaScript, Perl, etc.) the instrumentation code can be added to the interpreter or just-in-time (JIT) compiler (e.g., [Haldar et al. 2005; Vogt et al. 2007]). In this way, all the bookkeeping information the JIT needs for execution (e.g., type of variables, size of structures) is accessible to the tracking environment as

well. In fact, Perl has built in support for a taint mode [Perl Taint] which is enabled as soon as the real and effective group or user-IDS differ. This allows the interpreter to prevent a variety of common security flaws from being exploited.

To track the information flow on a binary level, the instrumentation is usually done in an environment where there is full control over the process under analysis. Such an environment can be based on dynamic binary instrumentation [Newsome and Song 2005], a full system emulator (e.g., [Chow et al. 2004; Egele et al. 2007; Portokalidis et al. 2006; Yin et al. 2007]), or even a hardware implementation [Crandall and Chong 2004] or [Venkataramani et al. 2008].

The value and applicability of information flow tracking for malware analysis is discussed by Cavallaro et al. [2008]. Researchers so far have demonstrated systems that perform data-, address-, and control flow tracking [Kim et al. 2009; Nanda et al. 2007]. Trishul as presented in Nair et al. [2008] even addresses the implicit information flows. However, these systems do not intend to be used for malware analysis. If information flow techniques become prevalent in analysis tools, it can be expected that malware authors will come up with mechanisms that try to circumvent such systems.

The feasibility of systems that perform address or pointer tainting is evaluated in Slowinska and Bos [2009]. The critiques raised therein apply to systems in which taint labels are propagated unrestricted on the physical level of an emulated system without leveraging higher-level information, such as process or library boundaries. However, some of the tools presented in Section 5 suggest address tainting is viable in an environment that uses this kind of information to limit taint propagation to those parts of the system that are deemed relevant for the analysis.

3.4. Instruction Trace

A valuable source of information for an analyst to understand the behavior of an analyzed sample is the instruction trace. That is, the sequence of machine instructions that the sample executed while it was analyzed. While commonly cumbersome to read and interpret, this trace may contain important information not represented in a higher-level abstraction (e.g., analysis report of system and function calls).

3.5. Autostart Extensibility Points

Autostart extensibility points (ASEPs [Wang et al. 2004]) define mechanisms in the system that allow programs to be automatically invoked upon the operating system boot process or when an application is launched. Most malware components try to persist during reboots of an infected host by adding themselves to one of the available ASEPs. It is, therefore, of interest to an analyst to monitor such ASEPs when an unknown sample is analyzed.

4. IMPLEMENTATION STRATEGIES

The implementation of a malware analysis system requires different design decisions that have far-reaching consequences. The different CPU privilege levels, as introduced in Section 3.1, imply that an analysis component that executes in a higher privilege level than the program under analysis cannot be accessed by this program. Analysis components executing on the same privilege level as the malware need to apply stealth techniques to remain hidden from the analyzed program. Implementing the analysis functionality in an emulator or virtual machine potentially allows an analysis approach to hide its presence even from malware that executes in kernel space. Of course, malware executing in a higher privilege level than the analysis component can always hide itself, and thus thwart being analyzed. This section describes the different possibilities of implementing an analysis system and explains the benefits and drawbacks of each method.

4.1. Analysis in User/Kernel Space

Analyzing malicious software in user space enables an analysis approach to gather information, such as invoked functions or API calls. Such an approach can easily access all memory structures and high-level information provided by the operating system. For example, enumerating the running processes is as trivial as querying the loaded modules or libraries. This is possible because the same set of APIs is available to the analysis tool as to every other application running on the system. The possibility of hiding the analysis component when only executing in user space is very limited. For example, hiding a process or a loaded library from all other processes running on the system is usually not possible from user space alone.

This limitation is eased when the analysis component runs in kernel space. Analysis tools with access to kernel-level functions can gather additional information, such as invoked system calls, and can hide its presence from malware that only executes in user mode. Hiding from malicious software that gained the privilege to run in kernel mode is, again, complicated.

Granted that the analysis component executes in kernel mode, API hooking is trivially achieved. However, it is somewhat more complicated to record an instruction trace or perform information flow tracing. One possibility of recording an instruction trace is to leverage the debugging facilities of the CPU to single step through the sample under analysis. Setting the trap flag in the x86 EFLAGS register accomplishes this task. Setting this flag throws in a debug exception for the next instruction. The analysis component can catch this debug exception and perform the necessary steps, such as disassembling the instruction and appending it to the instruction trace.

4.2. Analysis in an Emulator

Executing a program in an emulated environment allows an analysis component to control every aspect of program execution. Depending on which part of the execution environment is emulated, different forms of analysis are possible.

Memory and CPU Emulation. Emulating the CPU and memory results in a sandbox allows one to run potentially—malicious code without fearing negative impacts on the real system. A binary is executed in this sandbox by successively reading its instructions and performing equivalent operations in the virtual emulated environment. All side effects the binary produces are contained in the sandbox. To successfully emulate samples that rely on operating system services or shared libraries, these mechanisms need to be emulated as well. While projects exist that implement such emulation approaches (e.g., libemu [Baecher and Koetter]), the most prominent use of this technique is in engines of modern anti-virus products. Many AV suites employ CPU and memory emulation to overcome the hurdles imposed by packed or obfuscated binaries. To analyze a possibly packed binary, it is executed in the emulated environment. If the AV engine detects an unpacking sequence, the signatures are matched on the unencrypted memory contents contained in the emulated memory.

For a malicious sample to detect that it is running in such an emulated environment, it must perform an operation that requires components that are not or insufficiently emulated by the system. For example, the emulator might behave differently from a real CPU in the case of a CPU bug (if this bug is not considered in the emulation).

Full System Emulation. A full system emulator (e.g., [Bellard; Bochs] etc.) provides the functionality of a real computer system, including all required peripherals. In addition to CPU and memory, emulated devices, such as network interface cards and mass storage, are available. This setup makes it possible to install a common off-the-shelf operating system in the emulator. The OS that runs in the emulator is referred to as the *guest*, while the computer executing the emulator itself is called the *host*. An

emulator allows the host and guest architectures to be different. Thus, with the help of an emulator, it is possible, for example, to analyze software that is compiled for the ARM instruction set on a commodity x86 desktop computer. Implementing the analysis component as part of the emulator allows the analysis to be performed in a stealthy manner, and provides the possibility of monitoring the guest OS and all applications executed within. Since the emulator, and thus the analysis component as well, has full control of what the guest operating system perceives as its environment, it is possible for the analysis component to stay undetected. However, a malware sample can still detect side effects of the emulation or characteristics of the analysis environment and cease to work in such cases. For example, detecting imperfect emulation of CPU features allows a malware sample to recognize that it is running in an emulator. Comparing system properties, such as the currently logged-in user, to known values may allow a malware instance to detect a specific analysis tool. For a more detailed analysis of such "anti-analysis" techniques, see Section 4.6.2. Performing the analysis takes time resulting in a delay or slowdown in the guest operating system. This may open another possible avenue of detection for malicious samples.

Along with the benefit of having full control over the environment introduced by using an emulator comes the drawback of the *semantic gap*. An analysis approach implemented in an emulator has full access to the emulated system on the physical level. Nevertheless, the desired information consists of higher-level abstract concepts of the guest operating system (e.g., system calls, file handles, etc.). Thus, to access this information, the analysis tool must infer the high-level information from the raw system information by interpreting CPU state and memory contents similar to the guest OS. A common requirement for analysis systems that operate via full system emulation is that the analysis can be confined to a single process (i.e., the sample under analysis), and therefore omit the other components of the system from being analyzed. Here, the semantic gap manifests itself through the observation that on a hardware level, there is no concept of a process. However on x86 architectures, the page table base (CR3) register is unique for each process. Thus, by monitoring the APIs that create new processes, an analysis system can correlate the analyzed sample with a given CR3 register value. Monitoring APIs in such a system can be accomplished by comparing the value of the instruction pointer to the known function entry points in the dynamic libraries that provide the APIs. Care must be taken that the loader of the guest operating system may map a library at a different base address (than the preferred.) Thus, this list must be dynamically maintained.

Knowing the CR3 register value that corresponds to the process under analysis, it is trivial to produce an instruction trace for this process. In brief, every instruction that is executed while the CR3 register holds this value is part of the analyzed program. To lower the impact of the semantic gap, hybrid approaches exist that perform the analysis in the emulator and gather higher-level information (e.g., the list of loaded modules, etc.) with the help of an additional component running in the guest system [Bayer et al. 2006].

4.3. Analysis in a Virtual Machine

According to Goldberg [1974], a virtual machine (VM) is "an efficient, isolated duplicate of the real machine." A virtual machine monitor (VMM) is responsible for presenting and managing this duplicate to programs and is in charge of the underlying hardware. This means, in practice, that no virtual machine can access the hardware before the VMM assigns the hardware to this specific VM.

The key idea behind virtualization is that the privileged state of the physical machine is not directly accessible in any VM. The notion of privileged state observable from within the VM exists only virtually and is managed by the VMM to match the

guest system's expectations. In contrast to emulators, the host and guest architecture in virtual machines are identical. This restriction allows the guest system to execute nonprivileged instructions (i.e., instructions that do not affect the privileged state of the physical machine) unmodified on the host hardware, resulting in improved performance. Similarly as with emulators, however, virtual machines provide strong isolation for resources. Thus, an analysis component implemented in a VMM also has the potential to go unnoticed by the analyzed programs. Commonly, such an analysis component is either integrated directly into the VMM or takes place in an additional virtual machine. In the latter case, the VMM has to provide all necessary information to the analysis VM to perform its task, resulting in lower performance. Just as with an emulator, the VMM only has access to the state of the virtual machine. The semantic gap between this state and the high-level concepts of operating system objects must be bridged in order to retrieve the relevant information.

While it is trivial to create an instruction trace in an emulator, producing the same result on a VM-based analysis system is not so straightforward, because unprivileged instructions are executed on bare hardware without involving the VMM. Nevertheless, to produce an instruction trace, the same approach as with kernel-space analysis may be applied. Once the VMM learns the CR3 value of the analyzed process, it sets the trap flag in the CPUs EFLAGS register. The only difference is with environments that perform the analysis in the VMM; the debug exception would be caught in the VMM before it even reaches the guest operating system. To monitor API calls, the same restrictions as with emulation-based analysis techniques apply.

4.4. Resetting of the Analysis Environment

Another matter that may influence the design of an analysis approach is the amount of time it takes to reset the analysis environment to a clean state. This is necessary because results are only comparable if each sample is executed in an identical environment. The more samples that need to be analyzed, the bigger the impact of this reset time. So far, three methods have been proposed to perform this task: (1) software solutions, (2) virtual machine snapshots, (3) and hardware snapshots. The general idea behind the latter two techniques is to keep a "base system" (e.g., a plain installation of an operating system), and redirect any modifications to that system transparently to separate storage. Read operations, in turn, use that storage as an overlay to the original base system to correctly reflect the performed changes. A reset happens by omitting or creating a new empty overlay.

Software solution. The straightforward software approach consists of taking an image of the hard drive containing the base image with the analysis environment. After each analysis run, this image is restored from a tamper-proof clean-operating system, such as a Linux live CD.

Virtual machine snapshots. Most VMMs (and full system emulators) [VMWare snapshots ; Liguori 2010] provide mechanisms to take snapshots of the state of a virtual machine (i.e., virtual mass storage, CPU, and memory contents). Such systems rely on regular files as storage for the overlay data. VMMs and full system emulators commonly support taking and loading snapshots of a running system. This mechanism can be used to reduce the time required to analyze a sample, since no reboot is necessary to recreate a clean instance of the analysis environment.

Hardware support for snapshots. Similar to snapshots in VMMs, hardware solutions exist that redirect all changes directed to a base image to another physical drive or designated working area [Juzt-Reboot Technology]. Again, resetting this system is accomplished by omitting the overlay for read accesses. However, since sudden changes

to the underlying file system are unexpected to the guest operating system, such techniques require that a system is rebooted between analysis runs.

4.5. Network Simulation

Modern malware samples frequently require some form of Internet access to operate. For example, a malware sample might download additional components, updates, or configuration data before performing its nefarious actions. The trivial approach of denying or allowing all-network traffic usually yields undesired results. Denying all-network access to the sample under analysis will most likely result in incomplete observations of the malware's behavior. For example, a spam sending worm that cannot resolve the IP address of the mail server cannot even try to send spam emails. Thus, this characteristic behavior cannot be monitored. If all-network access is allowed, however, the same sample may participate in real spam campaigns which resembles unacceptable behavior. In addition to providing Internet access to a malware sample, it is also advantageous to provide easy targets for infection in case the malware attempts to spread over the network [Inoue et al. 2008]. Such targets are commonly provided by honeypots that imitate vulnerable network services. In this section, we discuss two methods to restrict network access for samples under analysis. These restrictions can be performed with different techniques and to varying extents.

No Internet, but simulated network. This approach does not allow the sample under analysis to communicate with the real Internet. Instead, commonly used network services are simulated, and all traffic is redirected to these services. Such services include but are not limited to DNS, mail, IRC, and file servers. If the simulation is sufficiently complete, the malware will exhibit its malicious behavior and the analysis is completely self-contained. Malware that tries to update over the Internet will fail to do so, and bots that wait for commands from their master most likely will stay dormant.

Filtered Internet access. While the malware is granted access to the Internet, this access is limited and tightly monitored. This is commonly accomplished by applying techniques that mitigate the malicious effect and volume of the malware generated traffic. Filtering outgoing mail traffic or applying intrusion-detection and prevention tools provide this functionality. By additionally applying traffic shaping or rate limiting on the malicious traffic, the negative effect on the Internet and its users is kept to a minimum.

4.6. The Malware Analysis Arms Race and Its Consequences

As mentioned in Section 2.3, an arms race has evolved between the authors of malware and security analysts who need to understand a malware's behavior in order to create efficient countermeasures. This section elaborates on the prevalent measures that malware authors apply today to allow their samples to evade analysis. Where appropriate, we also mention techniques that were subsequently adopted by security analysts and tools to overcome these countermeasures. Note that the use of these techniques is often not unique to malware. Authors of legitimate, benign programs may also make use of such techniques to protect their programs from being analyzed or reverse engineered.

4.6.1. Self-Modifying Code and Packers. Historically, malware used self-modifying code to make static analysis more cumbersome and disguise its malicious intents. While such modifications were first performed by incorporating the self-modifying parts in the malware itself, more recent developments have led to packer tools. A packer program automatically transforms an executable (e.g., a malware binary) into a syntactically different, but semantically equivalent, representation. The packer creates the semantically equivalent representation by obfuscating or encrypting the original binary and stores the result as data in a new executable. An unpacker routine is prepended to

the data, whose responsibility upon invocation lies in restoring (i.e., deobfuscating or decrypting) the data to the original representation. This reconstruction takes place solely in memory which prevents leaking any unpacked versions of the binary to the disk. After unpacking, the control is handed over to the, now unpacked, original binary that performs the intended tasks. Polymorphic variants of a given binary can be automatically created by choosing random keys for the encryption. However, their unpacking routines are, apart from the decryption keys, largely identical. Therefore, while signatures cannot assess the threat of the packed binary, signature matching can be used to detect the fact that a packer program was used to create the binary. Metamorphic variants, in contrast to polymorphic binaries, can also mutate the unpacking routine, and may encumber detection even more. According to Taha [2007] and Yan et al. [2008], a large percentage of malicious software today comes in packed form. Moreover, malware instances that apply multiple recursive layers of packers are becoming more prevalent.

4.6.1.1. *Mitigating the Packer Problem.* Dynamically analyzing packed binaries is, in theory, no different from analyzing nonpacked binaries, because once the unpacking routine is finished, the original binary is executed and behaves as if it were unmodified. However, in performing analysis that focuses solely on the original binary, it is advantageous to first undo the packing. Different approaches exist that allow reverting the modifications a packer has performed on a binary.

Reverse-engineering the unpacking algorithm. Understanding how the unpacking routine restores the original binary in memory allows one to create a program (unpacking tool) that performs the same actions and dumps the resulting binary to disk, making it available for analysis. Prior to reverting the obfuscation, one has to determine the packing tool used to produce the binary in question. For example, the PEiD [PEiD] tool is designed to answer this question.

Generic detection and reconstruction of packed code. For binaries that are obfuscated with unknown packers, other reconstruction techniques have been proposed in literature. The packing and unpacking steps need to be transparent for the original binary so that, once unpacking is finished, the binary needs to be present in its original form. Security researchers used this observation to propose a technique that allow them to detect the end of an unpacking sequence. Then, it is known when an unmodified version of the original binary is present in the process address space. This detection can be accomplished by enforcing a "write xor execute" ($W \oplus X$) policy on the packed binaries' virtual memory pages. This technique is outlined in the following three step algorithm.

(1) Upon startup, all memory pages of the binary under analysis are marked as executable and read-only.
(2) Once the binary writes to a memory page, a page fault occurs. When catching this write permission page fault, the system modifies the page protection settings to be read/write-only (i.e., not executable or NX).
(3) As soon as the unpacking routine is completed, it transfers control to the unmodified binary. This will lead to another page fault, due to the pages' NX page-protection settings.

By catching these execute page faults, an analysis system is able to recognize the end of an unpacking routine. Furthermore, the instruction that raised the execution page fault indicates the program entry point of the original binary. Following this approach allows an analysis system to unwrap a packed binary, even if it is recursively packed with different packer programs. Such a system would repeat the algorithm as necessary.

However, by querying the page-protection settings, a malware sample can detect this approach. It is important, therefore, for a tool implementing this algorithm to disguise these modifications from an analyzed sample. Furthermore, Virtual memory systems allow a process to map the same physical memory page at multiple different virtual pages, each with its individual protection flags. This would allow a malware instance to write and execute a page without being detected. Therefore, a dynamic unpacking tool should employ countermeasures against such evasion attempts.

4.6.2. Detection of Analysis Environments. Whereas static analysis has the potential to cover all possible execution flows through a program, dynamic analysis suffers from the problem of incomplete path coverage. Because the extraction of information is collected while the program is executed, only the actually executed paths are taken into account. This leads to malware instances that try to detect analysis platforms, and upon detection, either terminate or exhibit nonmalicious behavior to circumvent analysis.

Chen et al. [2008] present a taxonomy of different artifacts that may be used by malware to detect whether it is executed in an instrumented environment (e.g., in a virtual machine, or attached to a debugger). According to the taxonomy, such artifacts can be found in the following four major areas.

(1) *Hardware.* Devices in virtual machines often can be fingerprinted easily (e.g., VMWare's "pcnet32" network interface adapter, or KVM's "QEMU HARDDISK" virtual hard disk).
(2) *Execution Environment.* refers to artifacts that are introduced to a monitored process' environment (e.g., debugger status flags as accessible by the `IsDebuggerPresent()` Windows API call, or the memory address of the interrupt descriptor table (IDT) in VMWare guests).
(3) *External Applications.* The presence of well-known monitoring applications, such as debuggers, or file-system and registry-monitoring tools, can also be an indication for an analysis system.
(4) *Behavioral.* The execution time of privileged instructions vary between real hosts and systems that are executed in a virtual environment. It is straightforward for a malware program to capture these timing differences.

While malware samples are known to detect specific analysis frameworks, more generic approaches exist that detecting whether a program is executing inside an emulated or virtualized environment [Ferrie 2007; Lau and Svajcer 2008; Rutkowska 2004; Zeltser 2006]. Raffetseder et al. [2007] propose multiple approaches to detecting an emulator by detecting differences in the behavior of emulated system when compared to real hardware. These approaches rely on CPU bugs, model specific registers, and differences in timing. In addition, Garfinkel et al. [2007] illustrate detection possibilities based on logical, resource, and timing discrepancies. Since some analysis techniques facilitate the trap flag in the EFLAGS register to create fine-grained (on the machine instruction level) analysis, malware samples can detect such approaches by reading the EFLAGS register and inspecting the specific bit within.

Countering Anti-Analysis Techniques. Currently, many malware samples cease to exhibit their intended malicious behavior when they detect an emulation or virtualization environment. This behavior is exploited by Chen et al. [2008] to propose protection mechanisms through the implementation of light-weight fake artifacts of such environments. Their evaluation showed that in many cases, such mechanisms are successful in deterring malware from performing its harmful actions. The growing acceptance and application of virtualization technology in the foreseeable future will reveal whether this behavior is still viable. Alternatively, malware authors may drop

such generic detection techniques in order not to preclude too many legitimate targets from infection.

In order to discover and mitigate such evasion attempts, two solutions seem possible. First, one can try to learn what differences between the analysis environment and a common computer system make the malware change its behavior. By eliminating the differences and rerunning the analysis, the malware will exhibit its malicious behavior. The second possibility is to modify the malware during execution, thus forcing different execution paths to be executed [Moser et al. 2007a; Brumley et al. 2007].

Existing analysis systems already provide a variety of countermeasures to prevent their detection. Depending on which parts of the system are modified to perform the monitoring, different techniques may be applied to hide the presence of the analysis tool.

For approaches that implement at least parts of their analysis capabilities in the same operating system instance as the analyzed sample, it is possible to hide these components by employing rootkit techniques. For example, to hide additional analysis processes from the analyzed sample, a rootkit component can filter the results of the API calls that return the list of currently loaded processes. Similar techniques may be applied to filter modules that would enable a malware sample to identify an analysis system by enumerating all loaded modules. However, malware that contains a kernel-mode component can circumvent such hiding techniques by accessing the corresponding memory structures directly, without invoking any API calls.

Analysis systems that modify the analyzed sample during execution in memory can conceal these modifications by storing a copy of the unmodified pages and marking the modified memory pages as "not present" in the page table. A sample might verify its own integrity by computing a hash value over all its memory pages. By computing this value for an unmodified version of the binary and storing the intended value in the binary itself, the sample could detect modifications to its memory pages by comparing the two values. However, since the modified pages are marked as "not present," a page fault will occur for every access by the sample. The page fault handler can be set up to report an unmodified version of the memory page in such instances. Thus, the integrity check will pass and the modifications performed by the analysis environment are successfully concealed.

As described in Section 4.6.1, modifying the page-protection settings is a viable option for detecting whether a packed binary has completed its unpacking stage. Of course, a malware sample might inspect the page-protection settings of its own memory pages, and thus detect such systems. However, there are multiple ways an analysis system based on this technique can evade detection. For example, in systems that directly modify the page table entries (PTE) of the running process, the modifications can be disguised by maintaining shadow copies of the involved page table structures. Furthermore, a system can mark the page table containing the PTEs, themselves, as "not present." Thus, if an analyzed process accesses a modified PTE, it would trigger a page fault, and the page-fault handler can return the shadow version of that entry to the sample. Systems that operate outside of the operating system executing the analyzed sample (e.g., in VMMs, emulators, or hypervisors) can maintain shadow copies of the page tables that are not accessible to the analyzed sample at all. Thus, no additional effort is required to hide these structures from being detected. Similarly, such systems would modify the access permissions to the shadow page tables and trap and handle the resulting page fault on the higher level of privilege (i.e., in the hypervisor or emulator). By not propagating these page faults to the guest operating system, the modifications are successfully hidden.

Additionally, analysis systems that rely on the use of machine-specific registers (MSR) might be detected if these MSRs are accessed by the sample. For example, if

the trap flag in the x86 EFLAGS register is set, the completion of the next instruction raises a trap exception and clears the flag. Thus, by repeatedly setting the flag in the corresponding interrupt handler, one can easily produce an instruction trace of the monitored process. However, since this flag is usually cleared during normal operation, its state needs to be concealed from the executed sample. Furthermore, the `SYSENTER_EIP_MSR` machine-specific-register is used to perform system calls on recent CPUs and operating systems. Modifying its contents allows an analysis tool to conveniently monitor the invocation of system calls. However, this modification must be hidden from the analyzed sample. Commonly, systems that modify MSRs can conceal these modifications by maintaining shadow versions of these registers, ensuring that the analyzed sample only has access to the shadow versions.

4.6.3. Logic Bombs. Instead of detecting the analysis framework, malware can also implement logic bombs to hide its malicious intent. For example, a malware might only exhibit its malicious tasks on a certain date.[1] If the system date on the analysis machine indicates a different day at the time the analysis is performed, the malicious behavior cannot be observed. Nevertheless, all infected machines would perform the malicious tasks on the intended day. Another form of logic bomb is the reliance on user input. For example, a malware may stay dormant until a certain number of keys has been pressed by the user. If the analysis tool does not emulate this user input, the malicious tasks are not performed and, thus, cannot be monitored. Conditional code obfuscation [Sharif et al. 2008] can be applied to further hinder analysis tools. Following this scheme, the binary representation of a program that depends on a trigger mechanism (e.g., a bot receiving a command), is replaced with an encrypted version thereof. The encryption key is chosen such that it is implicitly available when the code should be executed (e.g., the received command string), but not present in the program otherwise. For example, a conditional piece of code of a bot program might start scanning for new victims as soon as the bot receives a "scan" command. This code is then encrypted with the key "scan", and the check for whether the code should be executed compares the hash values instead of the plain text. Thus, the key is successfully removed from the binary but is present as soon as the bot is instructed to execute a "*scan*" action.

4.6.4. Analysis Performance. Another important issue for malware analysis systems is performance. Every CPU cycle spent on the analysis is missing somewhere else. This results in a usually noticeable slowdown of the analysis target. More importantly, the execution of analysis tasks may lead to the violation of timing constraints in the operating system that executes the sample under analysis. For example, if the extensive analysis that must be performed at program startup delays the creation of the new process for too long, the OS might consider the process unresponsive and kill it, therefore inhibiting a thorough analysis of the process in question.

To counter this, different techniques are applied to cloak the slowdown introduced by the analysis. Such techniques include patching the RDTSC timestamp counter register or slowing down time in an emulated or virtual environment. These approaches only work for the host running the analysis. If the sample under analysis interacts with other components in the network, applying such timing tricks may lead to unreliable networking behavior due to timeouts.

Throughput—that is, the number of analyzed samples per time unit—can also be used as a performance indicator for malware-analysis systems. In dynamic analysis, samples are executed in a controlled environment. This execution takes time.

[1]The infamous Michelangelo virus only executes on the 6th of March, the birthday of the Italian Renaissance painter.

Additionally, managing tasks such as setting up the environment and collecting analysis results consumes time as well. Summing up these time results in an upper-bound limit on the performance of a given tool. Most malware, once started, remains running as long as the infected operating system runs. Thus, analyzing the whole lifetime of a malware sample is unfeasible. The common approach to counter that is to terminate the analysis once a given timeout expires. The value of this timeout has great influence on the throughput of an automated malware analysis system. Nicter [Inoue et al. 2008], for example, is able to analyze 150–250 samples per day. With samples executing for 30 seconds (wall clock time), the remaining 22 hours a day are consumed by managing tasks, such as resetting the analysis environment. If an analysis approach is implemented with scalability in mind, the throughput can be raised by employing more hardware resources, such as additional systems performing the analysis or faster equipment.

5. MALWARE ANALYSIS TOOLS

The following section gives an overview of the existing approaches and tools that make use of the presented techniques to analyze unknown and potentially malicious software. For each tool, we give a brief summary describing which of the techniques from Section 3 are implemented. A discussion on the approaches, possible evasion techniques, and advantages over other approaches is given as well.

The analysis reports generated by the tools in this section give an analyst valuable insights into actions performed by a sample. These reports lay the foundation for a fast and detailed understanding of the sample. This knowledge is necessary for developing countermeasures in a timely manner.

5.1. Anubis

The analysis of unknown binaries (Anubis) project [Anubis] is based on TTAnalyze presented in Bayer et al. [2006]. Anubis executes the sample under analysis in an emulated environment consisting of a Windows XP operating system running as the guest in Qemu [Bellard 2005]. The analysis is performed by monitoring the invocation of Windows API functions, as well as system service calls to the Windows Native API. Additionally, the parameters passed to these functions are examined and tracked.

Since Anubis executes the analyzed sample in a complete Windows operating system, it is important to focus on the operations that are executed on behalf of the sample and omit operations that occur as normal behavior of the OS or other running processes. To this end, Anubis makes use of the knowledge that Windows assigns each running process its own page directory. The physical address of the page directory of the currently running process is always present in the CR3 CPU register. Thus, upon the invocation of the analyzed sample, Anubis records the value of the CR3 register and performs the analysis only for this process. By monitoring the APIs and system calls that are responsible for creating new processes, Anubis is able to monitor all processes that are created by the original sample.

The monitoring of function calls is based on comparing the instruction pointer of the emulated CPU with the known entry points of exported functions in shared libraries. To this end, Anubis manages a list of functions to monitor with their corresponding function-entry points. These addresses need to be determined dynamically as only their offset, relative to the beginning of the library (i.e., base address), is known in advance. The loader may decide to load the library at a different address than the preferred base address. Thus, the real entry point to a function is only known after the loader has mapped the library into the process' memory space. To accomplish this, Anubis tracks all invocations of the dynamic loader in the context of the sample under analysis.

Together with the invocation of these functions, Anubis also monitors their arguments by passing the arguments to call-back routines that perform the necessary analysis and tracking steps. For example, the usage of handles for file and registry operations, or network sockets is tracked, allowing Anubis to produce expressive reports of the related activities.

5.2. Multiple-Path Exploration

Automatic dynamic malware-analysis tools generate their reports based on a single execution trace of the sample under analysis. The use of logic bombs allows malware to only reveal its malicious behavior based on arbitrary constraints. For example, a malware sample could postpone its malicious activities until a certain date is reached or stop executing if necessary files cannot be found on the infected system. To overcome this shortcoming, Moser et al. [2007a] present a tool capable of exploring multiple execution paths for Windows binaries. This tool recognizes a *branching point* whenever a control-flow decision is based on data that originates outside the monitored process. This data can only be introduced to the process via system calls. Thus, a branching point is detected if a control-flow decision is based on a return value of a system call (e.g., the current system time). Every time such a situation occurs, the tool takes a snapshot of the running process that allows the the system to reset to this state. Execution is continued and after a timeout (or process termination), the system is reset to the recorded snapshot. Then, the value that is responsible for the control-flow decision is manipulated such that the control flow decision is inverted, resulting in the execution of the alternate path.

This approach extends Anubis (see Section 5.1) and applies dynamic taint tracking to analyze how data returned from system calls is manipulated and compared by the process under analysis. The system calls are responsible for introducing the taint-labels handle-file system and registry access, as well as network activities and date/time information. When manipulating a value upon resetting the system state, special care is taken by the system to update the value used in the corresponding compare instruction in a consistent manner. This means that not only the value directly involved in the comparison must be changed, but all other memory locations that depend on this value must be manipulated in a consistent manner to make the execution of alternative paths feasible. To achieve this, the system stores a set of memory locations for each branching point that depends on the compared value combined with a set of linear constraints describing these dependencies. During a reset, the set of constraints is evaluated by a constraint solver to produce the values that need to be substituted to force execution down the other path. If a dependency cannot be modeled as a linear constraint (e.g., a value and its hash value have nonlinear dependencies), the system is unable to update the memory locations in a consistent manner.

5.3. CWSandbox

Willems et al. [2007] created a tool called CWSandbox that executes the sample under analysis either natively or in a virtual Windows environment. The analysis functionality is implemented by hook functions that perform the monitoring on the API level. Additionally, monitoring of the system call interface is implemented. The system is designed to capture the behavior of malicious software samples with respect to file-system and registry manipulation, network communication, and operating system interaction. The virtual system is a full installation of a Win32-class operating system under which the sample under analysis is executed, together with the analysis components.

API hooking is performed by rewriting the sample under analysis as soon as it is loaded into memory. The applied hooking technique installs a monitoring function that can perform the analysis before and after every API call. To this end, the malware

process is started in a suspended state, meaning that the sample and all the libraries it depends on are loaded to memory but no execution has taken place yet. During the initialization, CWSandbox examines the exported API functions of all loaded libraries and installs the necessary hooks. This happens by backing up and replacing those instructions that are located at the the first five bytes of the API function with a nonconditional jump (JMP) instruction to the monitoring function. When the sample invokes the API function, the control flow is diverted to the hook which performs parameter analysis. After the analysis, the hook executes the backed-up instructions before continuing execution in the original API function. Once the API function returns, control is implicitly given to the hook function that can now post process and sanitize the results of the API call before handing it back to the analyzed sample. This process works for all libraries that are defined in the import table of the sample because they are automatically loaded at the process startup. However, explicit binding of DLLs, as performed by Windows' `LoadLibrary` API call, allows an application to dynamically load a library during runtime. Since malware could use this to circumvent analysis, the `LoadLibrary` API is hooked, as well, and performs the previously mentioned binary rewriting once the requested library is loaded into the process' memory space. In addition to API hooking, CWSandbox also monitors the system-call interface allowing the analysis of malware that uses system calls directly, in order to evade analysis.

The analysis tool itself consists of two major parts: a controlling process and a DLL that is injected into all processes that need to be monitored. The controlling process, which is in charge of launching the sample under analysis, receives all analysis output from the hook functions. In order to have easy access to the hook functions, they are compiled into a DLL which is injected into the address space of the sample upon startup. A malware instance may try to evade analysis by spawning another process that escapes the monitoring, but as the APIs that control process and threat creation are hooked, this is not a viable evasion strategy. These hooks will inject the analysis DLL and perform the required binary rewriting in every process that is created by the sample under analysis. The hooks in these processes will also report their analysis information to the controlling process.

A process can query the operating system for running processes, as well as for loaded libraries. Since this information would reveal the presence of the analysis framework (i.e., the controlling process, the injected library, and the communication channel between them), CWSandbox applies rootkit techniques to hide all system objects that could reveal the presence of the analysis framework from the process under analysis. To this end, the APIs used to query these system details are hooked as well. Once these functions return, the corresponding hooks sanitize the results (e.g., filtering the controlling process from the current process list, or removing the injected DLLs from the loaded modules list), therefore hiding their presence from potential malware samples.

The output of an analysis run is a report file that describes, from a high-level view, what actions have been performed by the sample during analysis. This report can be used by a human analyst to quickly understand the behavior of the sample, as well as the techniques that are applied to fulfill its task. Having this information generated quickly and automatically allows the analyst to focus attention on the samples that require deeper (manual) analysis over samples that exhibit already known behavior.

5.4. Norman Sandbox

The Norman Sandbox [2003] is a dynamic malware-analysis solution which executes the sample in a tightly controlled virtual environment that simulates a Windows operating system. This environment is used to simulate a host computer as well as an attached local area network and, to some extent, Internet connectivity. The core idea behind the Norman Sandbox is to replace all functionality required by an analyzed

sample with a simulated version thereof. The simulated system, thus, has to provide support for operating system-relevant mechanisms, such as memory protection and multithreading support. Moreover, all required APIs must be present to give the sample the fake impression that it is running on a real system. Because the malware is executed in a simulated system, packed or obfuscated executables do not hinder the analysis itself. As described in Section 4.6.1, a packed binary would simply perform the unpacking step and then continue executing the original program. However, to minimize the time spent in analysis, binaries that are obfuscated by a known packer program are unpacked prior to analysis.

Norman Sandbox focuses on the detection of worms that spread via email or P2P networks, as well as viruses that try to replicate over network shares. In addition, a generic malware-detection technique tries to capture other malicious software.

The Norman Sandbox provides a simulated environment to the sample under analysis consisting of custom-made versions of user-land APIs necessary for executing the sample. The functions providing these APIs are heavily instrumented with the corresponding analysis capabilities. Furthermore, to keep the simulation self-contained, these replacement APIs do not perform any interactions with the real system. Instead, the results of such API calls are created to allow the malware to continue execution (e.g., filling in the correct API function-return values). Bookkeeping takes place if required to thwart some detection techniques applied by malicious software. For example, a malware sample might try to detect the analysis tool by writing to a file and trying to read from that file later on to check if the stored information is still there. If the analysis tool does not provide the correct results to the read request, the malware can detect that it is being analyzed and will terminate without revealing its true malicious intents.

Special care is taken with respect to networking APIs. All networking requests issued by the sample under analysis are redirected to simulated components. If, for example, a sample intends to spread itself via email, it has to contact an SMTP server to send email. The connection attempt to TCP port 25 is detected, and instead of opening a connection to the real server, the connection is redirected to a simulated mail server. This is not deleted by the sample under analysis, and it will start sending the mail commands, including the malicious payload. An analogous approach is followed when a sample tries to write to a simulated network share or tries to resolve host names to IP addresses via DNS queries.

The authors claim that it is not a security problem to feed additional information into the simulated system; thus, a sample under analysis might be allowed to download files from the "real" Internet. Even with this option in place, the connection is not under control of the sample, but controlled by the analysis tool that performs the download on behalf of the malware and passes the result on to the sample. The allowed requests are rigorously filtered, allowing only file downloads, while dropping all other requests. The rationale behind this is to counter the spread of worms (e.g., Nimda, CodeRed [Moore et al. 2002]) that reproduce themselves by sending HTTP requests to Web servers.

Instrumenting the APIs enables effective function-call hooking and parameter monitoring. The observed behavior (e.g., function calls and arguments) are stored to a log file. Norman Sandbox also monitors autostart extensibility points (ASEPs) that may be used by malware instances to ensure persistence and automatic invocation after shutdown/reboot sequences.

5.5. Joebox

During the dynamic analysis of a potentially malicious sample, Joebox [Buehlmann and Liebchen] creates a log that contains high-level information of the performed actions regarding file-system, registry, and system activities. Joebox is specifically designed

to run on real hardware, not relying on any virtualization or emulation technique. The system is designed as a client-server model in which a single controller instance can coordinate multiple clients that are responsible for performing the analysis. Thus, it is straightforward to increase the throughput of the complete system by adding more analyzing clients to the system. All analysis data is collected by the controlling machine.

Joebox hooks user-mode API calls, as well as system-call invocations. Every library providing an API contains a dictionary of the function names it exports. The names are accompanied with a pointer to the implementation of these functions within the library. This directory, called the export address table (EAT), is queried by a process that wishes to call a function in that library. The SSDT (system service descriptor table) is a similar structure that maps the system-call numbers to their implementation via function pointers. Joebox implements hooking onto these structures to perform its analysis. Hooking EAT entries allows Joebox to monitor function invocations in libraries, whereas hooking the SSDT entries provides insight into how the user-mode/kernel-mode interface is used by the sample under analysis.

Binary rewriting of API functions in user-mode is applied once the library is loaded to memory. Since these modifications are usually easy to detect by a running process, a kernel-mode driver is responsible for cloaking the performed changes. To prevent detection, the driver installs a page-fault handler and marks the memory page containing the executable code of the hooked function as "not present." As soon as a process tries to read from that memory page to detect possible modifications, the page-fault handler is called for the missing page and returns a fake, unmodified version of the page to the application. To further disguise the presence of an analysis framework, Joebox uses the AutoIT [Bennett] toolkit to emulate user interaction with the machine during the analysis phase. The rationale behind this is that malware may stay dormant until user interaction takes place in order to evade less sophisticated automated analysis tools.

5.6. Ether: Malware Analysis via Hardware Virtualization Extensions

Dinaburg et al. [2008] propose a general transparent malware-analysis framework based on hardware virtualization. They motivate their work by stating that existent analysis tools suffer from detectability issues that allow malicious code to detect that it is being monitored. Ether's transparency property results from the fact that it is implemented in a (Xen) hypervisor residing in a higher privilege level than the monitored guest operating system. Ether has support for monitoring executed instructions, memory writes, and (in Windows XP environments) execution of system calls. Furthermore, Ether provides mechanisms for the Windows XP version to limit the analysis to a specific process only.

Instruction execution monitoring is implemented by setting the trap flag of the CPU when code in the guest system is executed. This results in a debug exception arising after every executed machine instruction. Thus, Ether is able to record and trace the instructions actually executed by the guest operating system or the monitored process, respectively. With this information, a human analyst has a very detailed representation of the actions performed by the analyzed system. Different malware samples already check for the presence of the trap flag when employing anti-debugging techniques [Mehta and Clowes 2003; Falliere 2007]. If the trap flag is detected, these malwares may disguise their malicious intent and circumvent analysis. To prevent this situation, Ether monitors and modifies all instructions that access the CPU flags, thus presenting the expected result to the guest OS. A shadow version of the trap flag is maintained by Ether, representing the guest operating systems' assumption of the flag's value. Queries from the guest system to get or set the trap flag are always evaluated on the shadow version of the flag. This is necessary as malware samples are actively

using the trap flag to perform their task (e.g., polymorphic decoding engines [Labir 2005; Daniloff 1997]).

Monitoring memory writes is accomplished by setting all page table entries to "read only". As soon as the guest OS performs a write operation, a page fault is raised. Ether checks if the reason for this fault lies in normal guest operation (e.g., the system tries to access a swapped-out page), passing the faults on in these cases. All other faults are handled by Ether, as they indicate memory writes by the guest OS and the necessary analysis steps can be performed. Since Ether only modifies the settings of the shadow page tables which are not visible to the guest OS, these changes cannot be observed by software running in the guest OS.

Under a Windows XP guest, OS Ether implements system-call monitoring as follows. In modern CPUs the `SYSENTER_EIP_MSR` register holds the memory address of executable code responsible for dispatching system calls. For an application to perform a system call, it is necessary to load the system call number and arguments to the specified registers and stack locations, and then to execute the `sysenter` instruction. Execution then continues at the address stored in the `SYSENTER_EIP_MSR` register. This code, executing in kernel-mode, is responsible for reading the system call number and parameters and invoking the respective system functions. Ether monitors system-call invocations by modifying the value of the `SYSENTER_EIP_MSR` register to point to an unmapped memory region. Thus, accessing this address results in a page fault. As soon as a page fault occurs for the specified address, the system is aware that a system call was invoked, restores the original value, and executes the system call. Once the system call is detected, Ether has full access to the memory of the guest OS, enabling the analysis of the parameters. After completion, the register is reset to the invalid address to catch the next system-call invocation. Access to the `SYSENTER_EIP_MSR` is completely mediated by Ether, making it possible to present the expected result to the guest, upon request.

This technique only works for system calls invoked via the `SYSENTER` instruction. Thus, Ether also needs to provide monitoring capabilities for the deprecated `INT 2E`-style of system-call invocations. Again, the value of the register pointing to the interrupt descriptor table is changed to an address that will result in a page fault upon access. Similar techniques previously described are applied here.

5.7. WiLDCAT

WiLDCAT is a framework for coarse- and fine-grained malware analysis. It consists of multiple components that implement stealth breakpoints (Vasudevan and Yerraballi [2005]), binary instrumentation (Vasudevan and Yerraballi [2004] and Vasudevan and Yerraballi [2006b]), and localized execution (Vasudevan and Yerraballi [2006a]).

The development of stealth breakpoints is motivated by the observation that many malware samples employ code verification or dynamic code generation, which renders software breakpoints useless. In addition, hardware breakpoints can be rendered ineffective by malware if the debugging registers are used for computations. VAMPiRE [Vasudevan and Yerraballi 2005] implements breakpoints by setting the not-present flag of the memory page containing the instruction on which to break. A page-fault handler is installed to catch the fault that is raised, as soon as any instruction of that page is executed. If the instruction that triggered the fault matches the memory location of the breakpoint, VAMPiRE stops execution. Combined with a debugger, this system may be used to single-step through the application from that point on. The system implements stealth techniques to stop malware from detecting easily that it is being analyzed. For example, VAMPiRE uses the CPU's trap flag in the EFLAGS register to implement single-stepping. Similarly to Ether's approach, a shadow copy of this register is used to present any querying application with the expected answer, thus,

hiding the true value of the EFLAGS register. Furthermore, a clock patch is applied to conceal the latency introduced by the page-fault handler.

Two components in the framework provide binary instrumentation:

SAKTHI [Vasudevan and Yerraballi 2004] implements binary instrumentation by rewriting a target function to redirect control to an instrumented version. This allows for pre- and postprocessing of arguments and results, and also allows the system to call the original target function.

SPiKE [Vasudevan and Yerraballi 2006b] also provides binary instrumentation, however, instead of target-function rewriting, it builds on top of VAMPiRE. The system inserts breakpoints at desired locations in the source code and maintains a table which relates a breakpoint to a instrument. Once the breakpoint triggers, the system redirects control flow to the instrument that can perform any preprocessing as necessary. If needed, the instrument can call the unmodified version of the target function and postprocess or sanitize any results after its return. SPiKE also allows redirection in DLL files by modifying the binary file on disk before it is loaded. During this modification, SPiKE adds the analysis code to the DLL and modifies entries in its export address table to point to the analysis functions, instead of the target functions. An application invoking the instrumented function calls the hook function that performs the desired analysis and then calls the original function.

The last component in the WiLDCAT framework is Cobra [Vasudevan and Yerraballi 2006a], a system that supports localized executions, which refers to a technique that breaks a given program stream into smaller basic blocks that are executed in turn. A basic block is a sequence of instructions that is either terminated with a control flow-modifying instruction (e.g., `call`, `jmp`) or when a certain maximum length is reached. After the execution of each basic block, the system invokes an analysis function that can perform the necessary tasks on a fine-grained level. Because creating basic blocks at runtime is costly and the instrumentation may not be necessary in all cases, Cobra allows the user to define overlay points. An overlay point is the address of an instruction upon whose execution the system switches to localized execution. Release points, respectively, are used to stop the localized execution again. Pairs of overlay and release points can be used to prevent the system from performing localized execution in well known API functions. Apart from these exceptions, the premise of Cobra is to only execute code that is split into blocks and instrumented accordingly. Privileged instructions may be used by a malware sample to detect the analysis framework. To prevent this, the set of relevant privileged instructions has been identified, and the risk of detection is mitigated by scanning the basic blocks for occurrences of such functions and replacing these privileged instructions with stealth implants that hide all traces of the analysis tool.

5.8. Hookfinder

When malware gathers information from an infected system, it is important from an attacker's point of view that this happens in a stealthy manner, in order to evade detection. Commonly, malicious programs, such as spyware or rootkits, implant hooks into the system to be notified when events of their interest occur. For example, a keylogger for the Windows environment might create a hook using the `SetWindowHookEx` API function, which is notified whenever a key is pressed. Malware samples are free to employ any of the hooking techniques outlined in Section 3.1. Hookfinder is a system capable of detecting such hooking techniques and produces detailed reports on where these hooks are and how they were implanted into the system. The system monitors a process and detects an implemented hook if it observes that the control flow is

redirected to this process by an earlier modification to the system state (e.g., memory write, API calls, etc.).

Hookfinder implements this approach by employing data and address taint-tracking techniques in a modified version of the Qemu [Bellard 2005] full-system emulator. All memory locations written by the process under analysis are tainted, and their taint status is propagated through the system. Whenever the instruction pointer contains a tainted value, this is an indicator for a hook being executed. In addition to taint-tracking, Hookfinder also keeps a detailed impact trace. This trace comprises all instructions that involve tainted data and enables the reconstruction of the method used to implant the hook into the system. Based on the impact trace, Hookfinder can produce information, such as the name of the function used to implant the hook (e.g., `SetWindowHookEx`). Moreover, Hookfinder provides detailed information on how the malware calculates the address of a function pointer to overwrite. Such information includes querying of global data structures and traversing their members until the corresponding address is found. Hookfinder does not rely on any a priori knowledge of how hooks are implanted into a system, thus it is able to give detailed information on hooking techniques that have not been previously encountered.

Similarly as with Panorama (see Section 5.10.3), Hookfinder employs a kernel module that executes in the emulated operating system to bridge the semantic gap. This module is responsible for gathering the necessary high-level information (e.g., loaded libraries, currently executing process, etc.) and communicating this information to the underlying analysis system.

5.9. Dealing with Packed Binaries

The hurdles for packed binary analysis are outlined in Section 4.6.1. This section presents tools that implement techniques to reverse or mitigate these problems.

5.9.1. Justin. Guo et al. [2008] present Justin, an automatic solution to the packer problem. The main goal of Justin is to reverse the packing of malware to a state in which a common signature-based AV engine can detect the threat. Justin is based on the idea that after the unpacker routine is completed, a copy of the original malware is present in the memory space of the process. To apply a signature-based AV scanner successfully, two prerequisites must be fulfilled: (1) The address space layout of the program embedded within a packed binary after it is unpacked is the same as that of the program if it is directly loaded into memory, and (2) the unpacker completely unpacks the embedded program before transferring control to it. Requirement (1) results from AV scanners that rely on the memory layout of a binary and is usually fulfilled by unpacking routines, since most original executables rely on a nonchanging memory layout (i.e., they are not location independent). Requirement (2) is also fulfilled by most unpacking routines. Intuitively, consider a packer program whose unpacking routine does not unpack the whole original binary at once. This would require that, at the time the packed binary is generated, the packer has to disassemble the target binary and insert instructions that call the unpacker routine upon execution. If such an instruction is not generated at an instruction boundary (i.e., partially overwriting existing instructions) the call will not be executed by the CPU. Thus, the rest of the binary will not be unpacked and execution will most likely terminate abnormally. Since this process is nontrivial to implement and not guaranteed to work with arbitrary binaries, packer programs today refrain from implementing such techniques. However, if malware authors created their binaries in a way that allowed a packer to perform such modifications, (e.g., providing callbacks to an unpacking routine) assumption (2) would not hold.

The main challenge for Justin is in recognizing the end of execution of the unpacker method, which is identical to detecting the start of execution of the original code. Three different methods are introduced for detecting the end of execution of the unpacker method. Since the original binary is reconstructed in memory before execution, one method is based on detecting the transfer of control flow to a dynamically created or modified memory page. Moreover, packed programs should not be aware that they were unpacked before execution. Thus, the second method is based on the assumption that the stack of an unpacked binary should be similar to the stack of a freshly loaded, not-packed version of the program. The third method is based on the assumption that a successful unpack leads to the correct setup of the command line arguments to the original program on the stack.

To detect the execution of dynamically created code, Justin applies the technique described in Section 4.6.1 (i.e., W $\oplus$ X page protections). Once this is detected, the memory image is scanned by the AV scanner to check for possible known threats in the unpacked binary. To circumvent any problems that might arise when the binary itself tries to modify page-protection settings, Justin records the changes but keeps its own settings in place. When the binary queries the protection flags, Justin reports the recorded information, if present.

5.9.2. Renovo. Kang et al. [2007] propose Renovo as a dynamic solution to the packer problem that automatically extracts the hidden code of a packed executable. The authors argue that it is an intrinsic feature of packed programs that the original code will be dynamically generated and executed, no matter what packing software is used. Renovo is able to reconstruct a detailed representation of the original binary (i.e., its binary code). Additionally, Renovo produces useful information, such as the original entry point.

To reconstruct the hidden code that is restored by an unpacker method, Renovo applies full-system emulation techniques to monitor the execution. In addition to managing the system memory of the emulated system, Renovo also manages a shadow memory that keeps track of memory-write accesses at the byte level. This shadow memory holds a flag for every byte, indicating whether or not the byte in memory was written. Initially, all flags in the shadow memory are cleared. Every write by the analyzed program flags the affected bytes as being dirty. If the currently executed instruction (pointed to by the instruction pointer) is marked as being dirty, the system identifies the execution of a dynamically generated instruction. By this time, the system has detected a hidden layer. The hidden code and data are identified as the written memory contents, and the original entry point is given by the value of the instruction pointer. The authors argue that malware may apply multiple layers of packers to further complicate the analysis. To overcome this, Renovo stores the gathered data and resets the shadow memory dirty bits after a hidden layer is discovered, repeating the whole process if necessary. The technique to detect the execution of dynamically generated code is similar to the technique applied by Justin and described in Section 4.6.1. However, keeping the information in a shadow memory in the emulator frees the authors of Renovo to disguise their system from malware that modifies and verifies page-protection settings.

Renovo is implemented in a whole-system emulator. As mentioned in Section 4.2, such systems only have access to the hardware state of the emulated system. Since Renovo aims at analyzing single processes, the semantic gap between the hardware view and the operating system concept of a process must be bridged. This task is performed by a kernel module installed in the emulated system, whose task is to identify newly created processes and the loading of libraries. Library loading is important as the module loader may decide to map the DLL to a memory area that contains dirty

tagged bytes, which are no longer used by the program. To rule out the false impression that the module contains dirty labeled instructions, the labels of the memory regions occupied by a module are cleared as soon as, they are loaded.

5.9.3. PolyUnpack. Royal et al. [2006] identify obfuscation techniques of unpack-executing malware that hamper the work of malware-detection tools. To counter such evasion attempts, they propose a system that automatically extracts hidden code from unpack-executing malware: PolyUnpack. This system follows a hybrid static-dynamic analysis approach to identify the execution of dynamically generated code. This detection is facilitated by the following two phase algorithm: First, the binary under-analysis is disassembled. The second step comprises of running the binary in a tightly monitored environment in which for every executed instruction, the corresponding instruction trace is calculated by performing in-memory disassembly. If an instruction sequence in memory is not present in the disassembled binary from the first step, this is an indication of dynamically generated code that is about to be executed.

PolyUnpack operates on Microsoft Windows executables and applies debugging techniques to provide a tightly monitored environment. The second step of the detection algorithm single-steps through the binary and the calculation of instruction sequences, and their comparison is performed at every instruction. Special care has been taken to deal with shared libraries. Since it is very likely that single-stepping through DLL code will result in instruction sequences that are not present in the original binary, calls to functions exported by DLLs are handled in a special manner. Upon loading a DLL into the process' memory space, PolyUnpack keeps track of the memory area occupied by that DLL. Once execution takes place in such a memory area, the return address is read from the stack and a breakpoint is set to that address. The program is resumed and single-stepping is used again, once the breakpoint triggers (i.e., the exported function returns).

The result of PolyUnpack is either a plain-text disassembly of the unpacked code, a binary dump of the code, or a complete executable version thereof.

5.9.4. OmniUnpack. Martignoni et al. [2007] designed a framework to reverse the packing of binaries and run an off-the-shelf anti-virus scanner on the recovered code. OmniUnpack is implemented as a solution for Windows XP, and the analysis focuses on a single-packed binary. The authors argue that the unpack routine of a packer creates the binary's original code in memory before executing it. Once this behavior has been observed by the system, an anti-virus scanner is used to check whether the newly generated code contains any known malicious patterns. Similarly as with Justin and Renovo, OmniUnpack detects the execution of dynamically generated code by implementing the detection technique described in Section 4.6.1, following the $W \oplus X$ memory page-protection scheme.

In contrast to the aforementioned tools, however, OmniUnpack tries to avoid executing of the anti-virus scanner at every detection of such an instance. To this end, OmniUnpack exploits the observation that malicious code can only affect the underlying operating system through the system-call interface. Therefore, OmniUnpack defines a set of "dangerous" system calls (e.g., registry/network/file-write operations, process creation, etc.). Once such a dangerous system call is invoked, the system examines the set of modified and executed pages and invokes the AV scanner on these memory areas. Enforcing the page-protection policy and managing the sets of written and written-executed pages is performed by a kernel-mode component. The gathered information is relayed to the AV engine running in user space as soon as a dangerous system call is invoked by the program.

The usage of an AV scanner for scanning memory regions of an executing process instead scanning files offline, introduces a number of restrictions on the used

signatures. For example, the signatures must characterize those parts of the malware that must be present in memory when the dangerous system call is invoked. In addition, the signature should characterize the unpacked (original) version of the malware.

5.10. Spyware Analysis

As described in Section 2.1, spyware refers to malware samples that gather sensitive information from an infected machine and leak this information back to the attacker. This section elaborates on analysis tools specifically tailored to analyze such spyware components.

5.10.1. Behavior-Based Spyware Detection. Kirda et al. [2006] propose a hybrid static-dynamic analysis system to identify spyware security breaches. The system focuses on spyware that comes as a plugin to the Microsoft Internet Explorer, either as a browser helper object (BHO) or as a toolbar. Collecting sensitive information from a Web browser makes sense for an attacker, as this is the application from which one can gather online banking account details to initiate fake transactions, or retrieve a list of visited webpages to deliver customized advertisements to lure users into insidious purchases. The preferred way for BHOs and toolbars to retrieve information from the browser is by subscribing to the browser's events. These events are fired, for instance, when the user navigates to a new URL by clicking on a link, or when a webpage is finished downloading and rendering. Other events indicate that a form has been submitted and allow for the collection of login credentials. Based on the observation that a spyware component must use system services to leak the gathered information out of the process (e.g., writing to a file or sending over a network connection), the authors propose the following detection approach.

A component is spyware if it (1) monitors user behavior by interacting with the Web browser, and (2) invokes Windows API calls that can potentially leak information about this behavior.

The hybrid analysis approach applies dynamic analysis techniques to identify the event handlers invoked as a reaction to browser events. It uses static analysis methods to analyze the handler routines themselves. The system is implemented as an application that provides all necessary interfaces to lure a BHO into believing that it is running inside Internet Explorer. Once the component registers the event handlers, a series of events are simulated, and the components reactions are monitored. Every invoked event handler is statically checked for whether it includes Windows API calls that allow information to be leaked to the attacker. If such a system call is identified, the component is classified as being malicious.

5.10.2. TQana. The system presented by Egele et al. [2007] is able to dynamically analyze spyware is installed as a browser plugin to Microsoft Internet Explorer. These browser helper objects (BHOs) are at the core of a large diversity of malicious software that compromises the security of a user's system. By extending the Qemu [Bellard 2005] full-system emulator with information-flow capabilities, TQana is capable of tracking the propagation of sensitive data through the system. Taint labels are introduced to the system through the Web browser's `Navigate` event, which fires whenever the browser is instructed to visit a new location by either clicking a link, selecting a bookmark, or entering a new URL into the address bar. In addition, the page contents retrieved by the Web browser are tainted. Implemented taint sinks cover-writes to the file system or registry, sending tainted data over the network and writing tainted data to shared memory regions. TQana applies taint tracking to data dependencies, address dependencies, as well as to certain control-flow dependencies. The implemented taint-tracking system associates labels with each tainted byte, thus, identifying its origin. Furthermore, a status flag is associated with each byte, indicating whether the byte

was touched by the component under analysis. This distinction is crucial, as illustrated by the following example. The host part of the entered URL is tainted data, but it does not pose a security risk if the operating system transmits this data as part of a DNS request to resolve the host name with the corresponding IP address. If the same information is transmitted by the component under analysis to a remote server, however, this should be flagged as a possible security breach. For this distinction to be made, the authors introduce the concept of "instructions executed on behalf of the BHO." An instruction is considered to be executed on behalf of the BHO if it is present in the BHO binary, created dynamically by the BHO, or part of a function invoked by such instructions. To determine whether a function was invoked by the BHO, the system keeps track of the call stack at all function calls. A `call` instruction from the BHO sets a limit on the stack pointer, and all instructions are regarded as executed on behalf of the BHO as long as that call frame is on the stack. Whenever the data is flagged as "processed by the BHO" reaches a taint sink, a possible security breach is logged.

In addition to taint tracking, the system also implements API and system-call hooking. Furthermore, TQana monitors calls in the component object model (COM) subsystem in Windows. All analysis tasks are performed solely from outside the emulated system, and, thus, have to bridge the semantic gap between the hardware-level view of the emulator and the operating systems-notion of processes, threads, and virtual memory. The hooking is implemented by monitoring the instruction pointer and comparing its value to the function entry points. Upon a match, the hook function is executed in the emulator and gathers the required information from the emulated system's memory. In order to extract information about processes and threads, the corresponding memory structures are parsed by the system. Similarly, the list of loaded modules for each process is retrieved by processing the respective data structures. During the analysis of a sample, an AutoIT [Bennett] script emulates user interaction in the guest system, such as visiting webpages and filling out Web forms.

By performing the taint tracking on physical memory, all running processes can be monitored simultaneously. This implicitly provides the means to track information transfers between processes via shared memory or other interprocess communication mechanisms.

Performing all required analysis steps outside of the emulated system allows TQana to perform stealthy monitoring, because, as opposed to Panorama (Section 5.10.3) and Renovo (Section 5.9.2), no kernel-mode component in the guest system is required. However, following this approach requires adaption to the emulated guest operating system. To read the necessary information from the emulated memory, the system has to retrieve in-memory structures of the guest system which are not guaranteed to remain stable across operating system versions or service packs.

5.10.3. Panorama. A system able to capture system-wide information flow is presented by Yin et al. [2007]. The resulting information should give an analyst a detailed view of how a tested program interacts with sensitive information, thus allowing the analyst to assess this program's intentions. To support this approach, the authors implemented a dynamic information-flow system that applies data and address tainting. The taint sources based on input from the keyboard, the network, or the hard disk are responsible for introducing the taint labels to the system. Panorama does not include dedicated taint sinks but creates a taint graph during analysis of the suspicious program. The system uses a modified version of the Qemu full-system emulator to perform the taint tracking. The created graph is enriched with high-level information such as processes that access the data or network connections that correspond to a certain packet payload.

In order to bridge the semantic gap between the high-level OS view and the hardware state accessible by the emulator, the following techniques are applied.

Panorama knows for each instruction which process is currently executing and which module (i.e., dynamic library) is the origin of this instruction. Similarly as with Renovo (Section 5.9.2), this is realized by a kernel-mode component running in the emulated environment that monitors process creation and library loading, and communicates this information out of the emulated guest operating system. The correlation between sectors of the emulated hard disk and the abstract notion of a file in the emulated system is realized by the forensic tool The Sleuth Kit [Carrier]. For network connections, the sent IP packet headers are examined and the correlation is established based on virtual connections (i.e., source/destination addresses and port/service numbers).

With this detailed information, Panorama constructs a taint graph during an analysis for the analyzed sample. The taint graph is a representation of the processes and modules that operate on the tainted data. During the analysis, a set of test cases are executed that introduce sensitive data to the system, and Panorama monitors the behavior of the component under analysis, as well as the rest of the system with regards to this data. These inputs include passwords, common text, and network traffic, among others. All the test cases are designed to feed the data as input to known benign processes, such as a text editor or form fields of a webpage. In many cases, there is little reason for a component to access this data at all.

The authors introduce three policies that are evaluated on the taint graph to determine whether the analyzed sample behaves maliciously.

(1) Anomalous information-access behavior defines that an analyzed component should not touch labeled data at all (e.g., login passwords).
(2) Anomalous information leakage behavior detines that while a browser plugin that accesses the URLs a user is surfing to is not malicious by itself, the leakage of this information (e.g., to disk or to a network connection) clearly manifests malicious behavior.
(3) Excessive information access behavior is exhibited by rootkits that hide files, since they have to check and sanitize file and directory listings every time such a listing is requested.

6. MALWARE CLUSTERING—BEYOND PLAIN ANALYSIS

The previous section presented existing dynamic-analysis tools that create reports of the analyzed sample. Such systems are necessary tools for malware analysts to grasp insight into malware behavior. However, the sheer number of new malware samples that reach anti-virus vendors everyday requires further automated approaches to limit the number of samples that require human analysis. Therefore, different approaches have been proposed to classify unknown samples into either known malware families or highlight those samples that exhibit unseen behavior, thus suggesting closer human analysis. This section gives an overview of the proposed approaches that may be used to perform this classification.

6.1. Malware Clustering Based on Anubis

As a response to the thousands of malware samples an anti-malware company receives everyday, Bayer et al. [2009] introduced a system that to effectively and automatically clusters large sets of malicious binaries based on their behavior. By comparing malware behaviors, an analyst can focus on new threats and limit the time invested in samples exhibiting already known behavior.

The proposed technique relies on Anubis to generate execution traces of all the samples. Furthermore, Anubis was extended with taint-propagation capabilities, to

make use of additional information sources. If a malware sample, for example, creates a file whose filename is dependent on the current system date, this dependence is recognized by the system. In addition, taint propagation provides the capabilities to detect characteristics such as reading a process's own executable and transmitting it over the network, as commonly exhibited by worms. Finally, the set of control-flow decisions that depend on tainted data is also recorded.

Once the execution traces are created, augmented with the taint information, a behavioral profile is extracted for each trace. The information available from the traces is interpreted in an object-centric way. The profile consists of operating system objects (e.g., files, processes, etc.) and corresponding operations (e.g., read, write, create, etc.). Compared to the sequence of system calls is stored in the execution traces, this representation is semantically richer, therefore enabling the system to unify semantically equivalent behavior better. For example, reading 256 bytes from a file at once is semantically equivalent to reading one byte, 256 times. However, the representation in the execution trace is considerably different. During creation of the behavioral profiles, the information recorded through taint propagation is used to describe whether objects depend on each other, as just described.

The inferred behavioral profiles are used as input for a clustering algorithm that combines profiles describing similar behavior into coherent clusters. To demonstrate the scalability of their approach, the authors evaluated their tool on a set of 75,000 malware samples that were clustered within three hours.

6.2. Behavioral Classification of Malware

Lee and Mody [2006] propose a system that divides a body of malicious software samples into clusters by applying machine-learning techniques on behavioral profiles of the samples. The execution of these samples take place in a tightly controlled virtual environment. A kernel-mode monitor records all system-call invocations, along with their arguments. The retrieved information about a sample's interaction with the system is recorded into a behavioral profile consisting of information regarding the sample's interaction with system resources, such as writing files, registry keys, or network activity. To measure the similarity between two profiles, the edit distance, between them, is calculated in which the cost of a transformation is defined in an operation-cost matrix. The authors then apply a k-medoids clustering approach to divide the body of malware samples into clusters that combine samples with similar behavioral profiles.

Once training is complete, a new and unknown sample is assigned to the cluster whose cluster medoid is closest to the sample (i.e., nearest neighbor).

6.3. Learning and Classification of Malware Behavior

Packer programs make it easy for attackers to create a plethora of malware instances that cannot be matched by classical signature-based anti-virus products. Thus, it is important to find other means of classifying an unknown malware sample. To this end, Rieck et al. [2008] present a system that uses the behavioral information contained in the analysis reports produced by CWSandbox (see Section 5.3). First, behavioral profiles of each known malware family are extracted. Second, machine-learning techniques are applied to derive classifiers from these profiles, allowing for grouping malware instances that share similar behavior.

To support this, the authors run a commercial anti-virus scanner on a large body of collected malware samples to obtain labels for the individual samples. For the samples that could be identified, these labels correspond to the families the samples belong to. Based on these labels and the behavioral profiles extracted from the CWSandbox reports, the authors trained support vector machines (SVM) to build classifiers for the individual families. A given SVM only states the probability at the analyzed sample

Table I. Comparison of General Malware Analysis Tools

	Anubis (5.1)	Multipath exp. (5.2)	Clustering (6.1)	CWSandbox (5.3)	Learning & Clas. (6.3)	Norman Sandbox (5.4)	Joebox (5.5)	Ether (5.6)	WiLDCat (5.7)	Hookfinder (5.8)	Justin (5.9.1)	Renovo (5.9.2)	PolyUnpack (5.9.3)	OmniUnpack (5.9.4)	Behav. Spyw. (5.10.1)	TQana (5.10.2)	Panorama (5.10.3)	Behav. Clas. (6.2)
Analysis implementation																		
User-mode component	○	○	○	●	●	○	●	○	○	○	●	○	●	●	●	○	○	●
Kernel-mode component	○	○	○	●	●	○	●	○	●	●	●	●	○	●	●	○	●	●
Virtual machine monitor	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○
Full-system emulation	●	●	●	○	○	○	○	○	○	●	○	●	○	○	○	●	●	○
Full-system simulation	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Analysis targets																		
Single process	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●
Spawned processes	●	○	●	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
All processes on a system	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	●	●	●
Complete operating system	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	○
Analysis support for																		
API calls	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
System calls	●	●	●	●	●	●	●	●	○	●	○	○	○	●	●	●	●	●
Function parameters	●	●	●	●	●	●	●	○	○	●	○	○	○	○	○	●	●	●
File-system operations	●	●	●	●	●	●	●	○	○	●	○	○	○	●	●	●	●	●
Process/thread creation	●	●	●	●	●	●	●	○	○	●	○	○	○	○	●	●	●	●
Registry operations	●	●	●	●	●	●	●	○	○	●	○	○	○	●	●	●	●	●
COM components	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	●	○	○
Detecting dyn. generated code	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○
Restoring packed binaries	○	○	○	○	○	○	○	○	○	○	●	●	●	●	○	○	○	○
W ⊕ X page protection	○	○	○	○	○	○	○	○	○	○	●	●	○	●	○	○	○	○
Multiple layers of packers	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○
Signature matching after unpacking	○	○	○	○	○	○	○	○	○	○	●	○	○	●	○	○	○	○
Instruction trace	○	●	●	○	○	○	○	●	●	●	○	●	●	○	○	○	●	○
Information-flow tracking	○	●	●	○	○	○	○	○	○	●	○	○	○	○	○	●	●	○
Multiple-path exploration	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
ASEP monitoring	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○	○
Networking support																		
Simulated network services	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○
Internet access (filtered)	●	●	●	●	●	●	●	●	●	●	●	●	●	○	●	●	●	●
Performs cloaking																		
Shadow EFLAGS register	○	○	○	○	○	○	○	●	●	○	○	○	○	○	○	○	○	○
Process hiding	○	○	○	●	●	○	●	○	●	○	○	○	○	○	○	○	○	○
Dynamic-module hiding	○	○	○	●	●	○	●	○	○	○	○	○	○	○	○	○	○	○
Disguise-modified memory pages	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	○	○	○
Disguise elapsed time	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○
Clustering Support	○	○	●	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●
Simulated User Interaction	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	●	○

belonging to a family. The system may be confronted with samples that do not belong to any known families or exhibit behaviors that are characteristic for multiple families. The decision process is structured such that it each sample to exactly one family. If the sample exhibits behaviors from more than one family, or none at all, the sample is classified as unknown, which suggests closer manual inspection.

7. CORRELATING TECHNIQUES AND TOOLS

This section illustrates which of the presented tools implement which analysis techniques introduced in Section 3, as summarized Table I. The numbers in parenthesis in

the column headers refer to the the sections in which the specific tool is discussed in detail, while the labels in the first column denote support for a given analysis technique. The remainder of this section elaborates on the semantics of the individual features listed in the table.

Analysis implementation. The most distinguishing property of the mentioned tools is on which privilege level the analysis is performed in comparison with the privilege level of the analyzed sample. Anubis, Ether, TQana, and Panorama are all implemented in an emulator or a virtual machine. Thus, these analysis engines run, by definition, in higher-privilege levels than the analysis subject. However, each of these tools must bridge the already mentioned semantic gap between the hardware-level view of the analysis engine and the guest operating system.

The tools that employ components executed on the same system as the analyzed sample, in contrast, can easily access all required high-level information. However, these systems do implement precaution mechanisms implemented in kernel-mode to avoid being detected by the samples they analyze. Inherently, this prevents these systems from effectively analyzing suspicious samples that contain kernel-mode components.

Analysis targets. Anubis, CWSandbox, and Panorama use Windows XP (SP2) as the guest operating system to execute the samples. No other operating system is currently supported. CWSandbox is designed to work with any OS of the Win32 family (Windows 2000, XP, and Vista). Joebox executes and analyzes the malicious sample in Windows Vista. We could not find any information regarding the Windows version the Norman Sandbox simulates. Ether can be used with any operating system that executes as a Xen guest. However, monitoring system calls and analyzing their parameters is only available for Windows XP guest systems.

System Calls and Windows API. All tools implement the monitoring of system-call invocations. Operating system-agnostic approaches, such as Ether, do not hook the Windows API since they lack the necessary information of where—or more precisely at what memory addresses—the API functions reside. However, by adding support for VM introspection techniques as proposed by Garfinkel and Rosenblum [2003], Ether could gain these capabilities as well. WiLDCAT has no built-in support for hooking system calls, but setting a breakpoint at the beginning of the relevant functions allows the analyst to monitor their execution. The same approach can be applied to monitor Windows API calls.

Function Call Parameters. Most of the surveyed tools analyze the parameters passed to the monitored functions and relate that information, where appropriate. For example, an open-write-close sequence involving a single file can be represented as a related action. Since parameter type and count of a given API may differ between operating system versions, there is no built-in support for this approach in WiLDCAT or Ether. Again, VM introspection could be used to add this analysis technique to Ether.

Information Flow Tracking. Anubis, more specifically the multipath and clustering extensions, Panorama, Hookfinder, and TQana, all perform information flow tracking. Panorama and TQana use this information to detect undesired leakage of sensitive information. Hookfinder uses the taint information to identify the techniques a malware sample applied to hook itself into the system. Anubis, on the other hand, uses these analysis results to perform behavioral clustering on malware samples.

Packed Binaries. All four systems are intended to unpack packed binaries are able to detect dynamically generated code. However, PolyUnpack is the only system that does not rely on the W $\oplus$ X page-protection algorithm. Furthermore, only Renovo supports the binaries that are recursively packed with different packer programs. Justin, and OmniUnpack are designed to perform a signature-based anti-virus scan once the original binary is restored.

Network Simulation. Standard network management techniques, such as network address translation, traffic shaping, or rate limiting can be enforced on all analysis systems. Anubis, for example, redirects all outgoing SMTP connections to a real mail server that is configured to only store the spam mails for later analysis. That is, the mails are not delivered to their intended recipients, thus the system does not contribute to spam campaigns when analyzing spam bots. Furthermore, Anubis rate limits all outgoing connections to prevent analyzed malware from participating in DOS attacks.

Norman Sandbox contains built-in simulated network services for email and file servers. All requests from the analyzed malware for such services are redirected to these simulated components. Furthermore, an analyzed sample is allowed to download files from the Internet (e.g., additional malware or updates). However, the transfer is performed by the underlying OS and the result is forwarded to the analyzed sample. All other Internet access is blocked.

Cloaking Techniques. Systems running on the same privilege level as the analyzed malware apply different levels of cloaking techniques to hide themselves from the sample under analysis. WiLDCAT implements a plethora of hiding techniques to cover the modifications it has performed on the system. It keeps a shadow copy of the trap flag in the EFLAGS register, as well as disguises its changes to the page-protection settings. Furthermore, a clock patch is applied to disguise the difference between the elapsed wall clock time and the elapsed analysis time. Joebox, as well as CWSandbox, employs rootkit techniques to hide its presence from the analyzed malware. To this end, they contain a kernel-mode component that filters the results of system and API calls that might reveal their presence (e.g., filtering the list of running processes, or loaded modules). In addition, Joebox uses a cloaking technique to disguise the modifications on the malware binary code to thwart detection by self-checking malware samples.

Clustering Support. Anubis and CWSandbox have been used to provide the behavioral profiles necessary to implement behavioral clustering of malware. The system presented in Section 6.2 also performs clustering of malware samples.

Simulated User Interaction. Panorama and TQana are designed to detect the leakage of sensitive information. Therefore, these systems need to provide such information (i.e., simulated user input) as stimuli to possible spyware components being analyzed. Joebox also implements this to thwart evasion attempts by malware that stays dormant in the case of no user interaction.

8. CONCLUSION

Before developing countermeasures against malicious software, it is important to understand how malware behaves and what evasion techniques it might implement. This article presented a comprehensive overview of the state-of-the-art analysis techniques as well as the tools that aid an analyst to quickly and in detail gain the required knowledge of a malware instance's behavior.

Ever-evolving evasion techniques (e.g., self-modifying code) employed by malicious software to thwart static analysis led to the development of dynamic analysis tools. Dynamic analysis refers to the process of executing a malicious sample and monitoring its behavior. Most dynamic analysis tools implement functionality that monitors which APIs are called by the sample under analysis, or which system calls are invoked. Invocations of APIs and system calls by software are necessary in order to interact with its environment. Analyzing the parameters passed to these API and system functions allows multiple function calls to be grouped semantically. Furthermore, several analysis tools provide the functionality to observe how sensitive data is processed and propagated in the system. This information serves as a clue to an analyst in understanding what kind of data is processed by a malware sample. As a result, an analyst can identify actions performed to fulfil a sample's nefarious tasks.

Automated dynamic analysis results in a report that describes the observed actions the malware has performed while under analysis. These reports can be compiled into behavioral profiles that can be clustered to combine samples with similar behavioral patterns into coherent groups (i.e., families). Furthermore, this information can be used to decide which new malware samples should be given priority for thorough analysis (i.e., manual inspection). In order to achieve this, behavioral profiles of new threats can be automatically created by an analysis tool and compared with the clusters. While samples with behavioral profiles near an existing cluster probably are a variation of the corresponding family, profiles that deviate considerably from all clusters likely pose a new threat worth analyzing in detail. This prioritization has become necessary as techniques such as polymorphic encodings or packed binaries allow attackers to release hundreds of new malware instances everyday. Although such samples might evade static signature matching, their similar behavior observed through dynamic analysis might reveal their affiliation with a given malware family.

The information an analyst gains from these analysis tools allow a clearer understanding of a malware instance's behavior, thus, laying the foundation for implementing countermeasures in a timely and appropriate manner.

REFERENCES

ANUBIS. Analysis of unknown binaries. http://anubis.iseclab.org. (Last accessed, 5/10.)

AVIRA PRESS CENTER. 2007. Avira warns: targeted malware attacks increasingly also threatening German companies. http://www.avira.com/en/security_news/targeted_attacks_threatening_companies.html. (Last accessed, 5/10.)

BACKES, M., KOPF, B., AND RYBALCHENKO, A. 2009. Automatic discovery and quantification of information leaks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, 141–153.

BAECHER, P. AND KOETTER, M. x86 shellcode detection and emulation. http://libemu.mwcollect.org/. (Last accessed, 5/10.)

BAYER, U., MILANI COMPARETTI, P., HLAUSCHEK, C., KRÜGEL, C., AND KIRDA, E. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*.

BAYER, U., MOSER, A., KRÜGEL, C., AND KIRDA, E. 2006. Dynamic analysis of malicious code. *J. Comput. Virology 2,* 1, 67–77.

BELLARD, F. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track of the USENIX Annual Technical Conference*.

BENNETT, J. AutoIt Script Home Page. http://www.autoitscript.com/. (Last accessed, 5/10.)

BOCHS. Bochs: The open source IA-32 emulation project. http://bochs.sourceforge.net/. (Last accessed, 5/10.)

BRUMLEY, D., HARTWIG, C., LIANG, Z., NEWSOME, J., POOSANKAM, P., SONG, D., AND YIN, H. 2007. Automatically identifying trigger-based behavior in malware. In *Botnet Analysis and Defense,* W. Lee et. al. Eds.

BUEHLMANN, S. AND LIEBCHEN, C. Joebox: a secure sandbox application for Windows to analyse the behaviour of malware. http://www.joebox.org/. (Last accessed, 5/10.)

CACHEDA, F. AND VIÑA, Á. 2001. Experiencies retrieving information in the World Wide Web. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC'01)*. IEEE Computer Society, 72–79.

CARRIER, B. The sleuth kit. http://www.sleuthkit.org/sleuthkit/. (Last accessed, 5/10.)

CAVALLARO, L., SAXENA, P., AND SEKAR, R. 2008. On the limits of information flow techniques for malware analysis and containment. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 143–163.

CHEN, H., DEAN, D., AND WAGNER, D. 2004. Model Checking One Million Lines of C Code. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS'04)*.

CHEN, H. AND WAGNER, D. 2002. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*. 235–244.

CHEN, X., ANDERSEN, J., MAO, Z., BAILEY, M., AND NAZARIO, J. 2008. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Proceedings of the IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN'08)*. 177–186.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th USENIX Security Symposium*.

CHRISTODORESCU, M., JHA, S., AND KRUEGEL, C. 2007. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 5–14.

CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th International Symposium on Microarchitecture*.

DAN GOODIN (THE REGISTER). 2008. SQL injection taints BusinessWeek.com. http://www.theregister.co.uk/2008/09/16/businessweek_hacked/. (Last accessed, 5/10.)

DANIEL, M., HONOROFF, J., AND MILLER, C. 2008. Engineering heap overflow exploits with javascript. In *Proceedings of the 2nd USENIX Workshop on Offensive Technologies (WOOT'08)*.

DANILOFF, I. 1997. Virus analysis 3, fighting talk. *Virus Bull. J.* 10–12.

DINABURG, A., ROYAL, P., SHARIF, M. I., AND LEE, W. 2008. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 51–62.

EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. X. 2007. Dynamic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference*. 233–246.

EGELE, M., SZYDLOWSKI, M., KIRDA, E., AND KRÜGEL, C. 2006. Using static program analysis to aid intrusion detection. In *Proceedings of the 3rd International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. 17–36.

FALLIERE, N. 2007. Windows anti-debug reference. http://www.symantec.com/connect/es/articles/windows-anti-debug-reference. (Last accessed, 5/10.)

FENG, H. H., GIFFIN, J. T., HUANG, Y., JHA, S., LEE, W., AND MILLER, B. P. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. 194 – 208.

FERRIE, P. 2007. Attacks on virtual machine emulators. www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf. (Last accessed, 5/10.)

FOSSI, M., JOHNSON, E., MACK, T., TURNER, D., BLACKBIRD, J., LOW, M. K., ADAMS, T., MCKINNEY, D., ENTWISLE, S., LAUCHT, M. P., WUEEST, C., WOOD, P., BLEAKEN, D., AHMAD, G., KEMP, D., AND SAMNANI, A. 2009. Symantec global Internet security threat report trends for 2008. http://www4.symantec.com/Vrt/wl?tu_id=gCGG123913789453640802. (Last accessed, 5/10.)

FREE SOFTWARE FOUNDATION. Code Gen Options - Using the GNU Compiler Collection (GCC). http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html#Code-Gen-Options. (Last accessed, 1/10.)

FRISK SOFTWARE INTERNATIONAL. 2003. F-prot virus signature updates cause false alarm in Windows 98. http://www.f-prot.com/news/vir_alert/falsepos_invictus.html. (Last accessed, 5/10.)

GARFINKEL, T., ADAMS, K., WARFIELD, A., AND FRANKLIN, J. 2007. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*.

GARFINKEL, T. AND ROSENBLUM, M. 2003. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*.

GOLDBERG, R. P. 1974. Survey of virtual machine research. *IEEE Comput. Mag.* June, 34–45.

GUO, F., FERRIE, P., AND TZI-CKER CHIUEH. 2008. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th International Symposium On Recent Advances In Intrusion Detection (RAID)*.

HALDAR, V., CHANDRA, D., AND FRANZ, M. 2005. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*. 303–311.

HUNT, G. AND BRUBACHER, D. 1999. Detours: binary interception of Win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*. USENIX Association, Berkeley, CA, 135–143.

INOUE, D., YOSHIOKA, K., ETO, M., HOSHIZAWA, Y., AND NAKAO, K. 2008. Malware behavior analysis in isolated miniature network for revealing malware's network activity. In *Proceedings of the IEEE International Conference on Communications (ICC)*.

JANSEN, B. J. AND SPINK, A. 2005. An analysis of web searching by european AlltheWeb.com users. *Info. Process. Manag. 41,* 2, 361–381.

JOHN LEYDEN (THE REGISTER). 2007. Kaspersky false alarm quarantines Windows Explorer. http://www.channelregister.co.uk/2007/12/20/kaspersky_false_alarm/. (Last accessed, 5/10.)

JUZT-REBOOT TECHNOLOGY. Juzt-reboot, intelligent back-up technology, instant recovery. http://www.juzt-reboot.com/. (Last accessed, 5/10.)

KANG, M. G., POOSANKAM, P., AND YIN, H. 2007. Renovo: a hidden code extractor for packed executables. In *Proceedings of the ACM Workshop on Recurring Malcode*. ACM Press, New York, NY, 46–53.

KANICH, C., KREIBICH, C., LEVCHENKO, K., ENRIGHT, B., VOELKER, G. M., PAXSON, V., AND SAVAGE, S. 2008. Spamalytics: an empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. 3–14.

KIM, H. C., KEROMYTIS, A. D., COVINGTON, M., AND SAHITA, R. 2009. Capturing information flow with concatenated dynamic taint analysis. In *Proceedings of the 1st International Conference on Availability, Reliability and Security*. 355–362.

KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. 2006. Subvirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy*. 314–327.

KIRDA, E., KRUEGEL, C., BANKS, G., VIGNA, G., AND KEMMERER, R. A. 2006. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*.

LABIR, E. 2005. Vx reversing III yellow fever (Griyo 29a). *CodeBreakers J. 2,* 1.

LAU, B. AND SVAJCER, V. 2008. Measuring virtual machine detection in malware using DSD tracer. *J. Comput. Virology*.

LEE, T. AND MODY, J. J. 2006. Behavioral classification. In *Proceedings of the European Institute for Computer Antivirus Research Conference (EICAR'06)*.

LIGUORI, A. 2010. Qemu snapshot mode. http://wiki.qemu.org/Manual. (Last accessed, 5/10.)

MARCUS, D., GREVE, P., MASIELLO, S., AND SCHAROUN, D. 2009. Mcafee threats report: Third quarter 2009. http://www.mcafee.com/us/local_content/reports/7315rpt_threat_1009.pdf. (Last accessed, 5/10.)

MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. 2007. Omniunpack: fast, generic, and safe unpacking of malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE Computer Society, Los Alamitos, CA, 431–441.

MEHTA, N. AND CLOWES, S. 2003. Shiva. advances in ELF binary runtime encryption. http://www.securereality.com.au/. (Last accessed, 5/10.)

MICROSOFT CORPORATION. 2006. Microsoft security bulletin MS06-014—Vulnerability in the microsoft data access components (MDAC) function could allow code execution. http://www.microsoft.com/technet/security/Bulletin/MS06-014.mspx. (Last accessed, May 2010.)

MICROSOFT CORPORATION. 2008. Microsoft security bulletin MS08-067 Critical; vulnerability in server service could allow remote code execution. http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx. (Last accessed, May 2010.)

MOORE, D., SHANNON, C., AND CLAFFY, K. C. 2002. Code-red: a case study on the spread and victims of an Internet worm. In *Proceedings of the Internet Measurement Workshop*. 273–284.

MOSER, A., KRUEGEL, C., AND KIRDA, E. 2007a. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.

MOSER, A., KRUEGEL, C., AND KIRDA, E. 2007b. Limits of static analysis for malware detection. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. 421–430.

NAIR, S. K., SIMPSON, P. N. D., CRISPO, B., AND TANENBAUM, A. S. 2008. A virtual machine-based information flow control system for policy enforcement. *Electron. Notes Theor. Comput. Sci. 197,* 1, 3–16.

NANDA, S., LAM, L.-C., AND CHIUEH, T.-C. 2007. Dynamic multi-process information flow tracking for web application security. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware Companion*. ACM Press, New York, NY, 1–20.

NEBBETT, G. 2000. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA.

NEWSOME, J. AND SONG, D. X. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*.

NORMAN SANDBOX. 2003. Norman SandBox Whitepaper. http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf. (Last accessed, 5/10.)

PEiD. PEiD: Packer Identification. http://www.peid.info/. (Last accessed, 5/10.)

PERL TAINT. Perl security / taint mode. http://perldoc.perl.org/perlsec.html#Taint-mode. (Last accessed, 5/10.)

PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. 2006. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In *Proceedings of the 1st EuroSys Conference*. 15–27.

PROVOS, N., MAVROMMATIS, P., RAJAB, M. A., AND MONROSE, F. 2008. All your iFRAMEs point to us. In *Proceedings of the 17th USENIX Security Symposium*.

PROVOS, N., MCNAMEE, D., MAVROMMATIS, P., WANG, K., AND MODADUGU, N. 2007. The ghost in the browser: Analysis of web-based malware. In *Proceedings of the 1st Workshop on Hot Topics in Understanding Botnets (HotBots'07)*.

RAFFETSEDER, T., KRÜGEL, C., AND KIRDA, E. 2007. Detecting system emulators. In *Proceedings of the 10th International Conference on Information Security (ISC'07)*. 1–18.

RIECK, K., HOLZ, T., WILLEMS, C., DÜSSEL, P., AND LASKOV, P. 2008. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 108–125.

ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. 2006. Polyunpack: automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*. 289–300.

RUTKOWSKA, J. 2004. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.invisiblethings.org/papers/redpill.html. (Last accessed, 5/10.)

RUTKOWSKA, J. 2006. Introducing Blue Pill. http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html. (Last accessed, 5/10.)

SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. 2008. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

SKOUDIS, E. AND ZELTSER, L. 2003. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ.

SLOWINSKA, A. AND BOS, H. 2009. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*. ACM Press, New York, NY, 61–74.

SOTIROV, A. Heap feng shui in javascript. http://www.phreedom.org/research/heap-feng-shui/heap-feng-shui.html. (Last accessed, 5/10.)

SPAFFORD, E. H. 1989. The Internet worm incident. In *Proceedings of the 2nd European Software Engineering Conference*. 446–468.

STASIUKONIS, S. 2007. Social engineering, the USB way. http://www.darkreading.com/security/perimeter/showArticle.jhtml?articleID=208803634. (Last accessed, 5/10.)

STONE-GROSS, B., COVA, M., CAVALLARO, L., GILBERT, B., SZYDLOWSKI, M., KEMMERER, R. A., KRUEGEL, C., AND VIGNA, G. 2009. Your botnet is my botnet: analysis of a botnet takeover. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS'09)*. 635–647.

SZOR, P. 2005. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional.

TAHA, G. 2007. Counterattacking the packers. http://www.mcafee.com/us/local_content/white_papers/threat_center/wp_counterattacking_packers.pdf. (Last accessed, 5/10.)

TANACHAIWIWAT, S. AND HELMY, A. 2006. Vaccine: War of the worms in wired and wireless networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communication Societies (INFOCom)*.

VASUDEVAN, A. AND YERRABALLI, R. 2004. Sakthi: A retargetable dynamic framework for binary instrumentation. In *Proceedings of the Hawaii International Conference in Computer Sciences*.

VASUDEVAN, A. AND YERRABALLI, R. 2005. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*. 381–392.

VASUDEVAN, A. AND YERRABALLI, R. 2006a. Cobra: fine-grained malware analysis using stealth localized-executions. In *Proceedings of the IEEE Symposium on Security and Privacy*. 264–279.

VASUDEVAN, A. AND YERRABALLI, R. 2006b. Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference*. 311–320.

VENKATARAMANI, G., DOUDALIS, I., SOLIHIN, Y., AND PRVULOVIC, M. 2008. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA'08.)*. 173–184.

VMWARE SNAPSHOTS. VMWare using snapshots. http://www.vmware.com/support/ws55/doc/ws_preserve_using_sshot.html. (Last accessed, 5/10.)

VOGT, P., NENTWICH, F., JOVANOVIC, N., KRUEGEL, C., KIRDA, E., AND VIGNA, G. 2007. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07)*.

WANG, Y.-M., ROUSSEV, R., VERBOWSKI, C., JOHNSON, A., WU, M.-W., HUANG, Y., AND KUO, S.-Y. 2004. Gatekeeper: Monitoring auto-start extensibility points (ASEPs) for spyware management. In *Proceedings of the 18th USENIX Conference on System Administration*. USENIX Association, Berkeley, CA, 33–46.

WILLEMS, C., HOLZ, T., AND FREILING, F. 2007. Toward automated dynamic malware analysis using CWSandbox. *IEEE Secur. Privacy 5,* 2, 32–39.

XU, J., SUNG, A. H., CHAVEZ, P., AND MUKKAMALA, S. 2004. Polymorphic malicious executable scanner by api sequence analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems*. 378–383.

YAN, W., ZHANG, Z., AND ANSARI, N. 2008. Revealing packed malware. *IEEE Secur. Privacy 6,* 5, 65–69.

YIN, H., SONG, D. X., EGELE, M., KRUEGEL, C., AND KIRDA, E. 2007. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 116–127.

ZELTSER, L. 2006. Virtual machine detection in malware via commercial tools. http://isc.sans.org/diary.html?storyid=1871. (Last accessed, 5/10.)

ZHUGE, J., HOLZ, T., SONG, C., GUO, J., HAN, X., AND ZOU, W. 2008. Studying malicious websites and the underground economy on the Chinese web. In *Proceedings of the 7th Workshop on Economics of Information Security*.

ZOVI, D. D. 2006. Hardware virtualization based rootkits. In *Proceedings of the Black Hat Briefings and Training Conference*.

Received June 2009; revised February 2010; accepted May 2010