A Survey of Stealth Malware Attacks, Mitigation Measures, and Steps Toward Autonomous Open World Solutions

Ethan M. Rudd, Andras Rozsa, Manuel Günther, and Terrance E. Boult

***Abstract*—As our professional, social, and financial existences become increasingly digitized and as our government, healthcare, and military infrastructures rely more on computer technologies, they present larger and more lucrative targets for malware. Stealth malware in particular poses an increased threat because it is specifically designed to evade detection mechanisms, spreading dormant, in the wild for extended periods of time, gathering sensitive information or positioning itself for a high-impact zero-day attack. Policing the growing attack surface requires the development of efficient anti-malware solutions with improved generalization to detect novel types of malware and resolve these occurrences with as little burden on human experts as possible. In this paper, we survey malicious stealth technologies as well as existing solutions for detecting and categorizing these countermeasures autonomously. While machine learning offers promising potential for increasingly autonomous solutions with improved generalization to new malware types, both at the network level and at the host level, our findings suggest that several flawed assumptions inherent to most recognition algorithms prevent a direct mapping between the stealth malware recognition problem and a machine learning solution. The most notable of these flawed assumptions is the *closed world* assumption: that no sample belonging to a class outside of a static training set will appear at query time. We present a formalized *adaptive open world framework* for stealth malware recognition and relate it mathematically to research from other machine learning domains.**

***Index Terms*—Stealth, malware, rootkits, intrusion detection, machine learning, open set, recognition, anomaly detection, outlier detection, extreme value theory, novelty detection.**

I. Introduction

MALWARES have canonically been lumped into categories such as viruses, worms, Trojans, rootkits, etc. Today's advanced malwares, however, often include many components with different functionalities. For example, the same malware might behave as a virus when spreading over a host, behave as a worm when propagating through a network, exhibit *botnet* behavior when communicating with command and control (C2) servers or synchronizing with other infected machines, and exhibit *rootkit* behavior when concealing itself from an intrusion detection system (IDS). A thorough study of all aspects of malware is important for developing security products and computer forensics solutions, but stealth components pose particularly difficult challenges. The ease or difficulty of repairative measures is irrelevant if the malware can evade detection in the first place.

While some authors refer to all stealth malwares as *rootkits*, the term rootkit properly refers to the modules that redirect code execution and subvert expected operating system functionalities for the purpose of maintaining stealth. With respect to this usage of the term, rootkits deviate from other stealth features such as elaborate code mutation engines that aim to change the appearance of malicious code so as to evade signature detection without changing the underlying functionality.

As malwares continue to increase in quantity and sophistication, solutions with improved generalization to previously unseen malware samples/types that also offer sufficient diagnostic information to resolve threats with as little human burden as possible are becoming increasingly desirable. Machine learning offers tremendous potential to aid in stealth malware intrusion recognition, but there are still serious disconnects between many machine learning based intrusion detection "solutions" presented by the research community and those actually fielded in IDS software. Sommer and Paxson [1] discuss several factors that contribute to this disconnect and suggest useful guidelines for applying machine learning in practical IDS settings. Although their suggestions are a good start, we contend that refinements must be made to machine learning algorithms themselves in order to effectively apply such algorithms to the recognition of stealth malware. Specifically, there are several flawed assumptions inherent to many algorithms that distort their mappings to realistic stealth malware intrusion recognition problems. The chief among these is the *closed-world* assumption – that only a fixed number of known classes that are available in the training set will be present at classification time.

Our contributions are as follows:

- *We present the first comprehensive academic survey of stealth malware technologies and countermeasures.* There have been several light and narrowly-scoped academic surveys on rootkits [2]–[4], and many broader surveys on the problem of intrusion detection, e.g., [5]–[7], some specifically discussing machine learning intrusion detection techniques [1], [8]–[10]. However, none of

Manuscript received February 19, 2016; revised August 21, 2016, September 16, 2016, and November 25, 2016; accepted December 1, 2016. Date of publication December 8, 2016; date of current version May 31, 2017. This work was supported in part by the National Science Foundation, NSF grant number IIS-1320956.

The authors are with the Vision and Security Technology Laboratory, Department of Computer Science, University of Colorado at Colorado Springs, Colorado Springs, CO 80918 USA (e-mail: erudd@vast.uccs.edu).

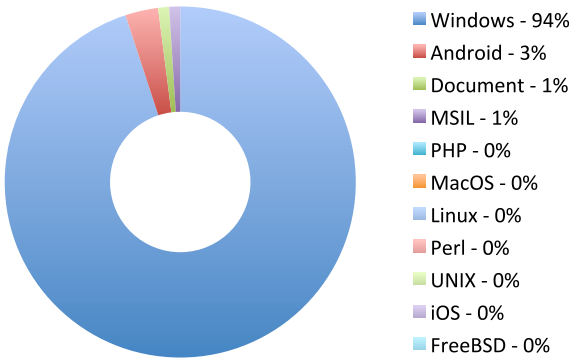
Digital Object Identifier 10.1109/COMST.2016.2636078

these works come close to addressing the mechanics of stealth malware and countermeasures with the level of technical and mathematical detail that we provide. Our survey is *broader in scope* and *more rigorous in detail* than any existing academic rootkit survey and provides not only detailed discussion of the mechanics of stealth malwares that goes far beyond rootkits, but an overview of countermeasures, with rigorous mathematical detail and examples for applied machine learning countermeasures.

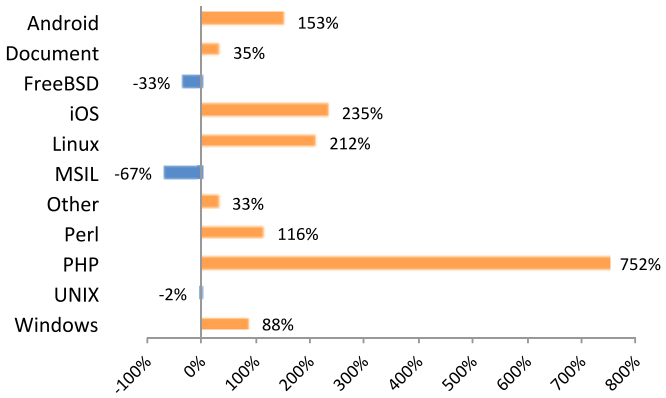
- *We analyze six flawed assumptions inherent to many machine learning algorithms* that hinder their application to stealth malware intrusion recognition and other IDS domains.
- *We propose an adaptive open world mathematical framework for stealth malware recognition* that obviates the six inappropriate assumptions. Mathematical proofs of relationships to other intrusion recognition algorithms/frameworks are included, and the formal treatment of open world recognition is mathematically generalized beyond previous work on the subject.

Throughout this work, we will mainly provide examples for the Microsoft Windows family of operating systems, supplementing where appropriate with examples from other OS types. Our primary rationale for this decision is that, according to numerous recent tech reports from anti-malware vendors and research groups [11]–[19], malware for the Windows family of operating systems is still far more prevalent than for any other OS type (see Fig. 1). Our secondary rationale is that within the academic literature that we examined, we found comparatively little research discussing Windows security. We believe that this gap needs to be filled. Note that many of the stealth malware attacks and defenses that apply to Windows have their respective analogs in other systems, but each system has its unique strengths and susceptibilities. This can be seen by comparing our survey to parts of [20], in which Faruki *et al.* provide a survey of generic Android security issues and defenses. Nonetheless, since our survey is about stealth malware, and not exclusively Windows stealth malware, we shall occasionally highlight techniques specific to other systems and mention discrepancies between systems. Unix/Linux rootkits shall also be discussed because the development of Unix rootkits pre-dates the development of Windows. Any system call will be marked in `a special font`, while proper nouns are highlighted differently. A complete list of Windows system calls discussed in this paper is given in Tab. I of the Appendix.

The remainder of this paper is structured as follows: In Section II we present the problems inherent to stealth malware by providing a comprehensive survey of stealth malware technologies, with an emphasis on rootkits and code obfuscation. In Section III, we discuss stealth malware countermeasures, which aim to protect the integrity of areas of systems known to be vulnerable to attacks. These include network intrusion recognition countermeasures as well as host intrusion recognition countermeasures. Our discussion highlights the need for these methods to be combined with more generic recognition techniques. In Section IV, we discuss some of these more generic stealth malware countermeasures in the research literature, many of which are based on machine learning. In Section V, we identify six critical flawed algorithmic assumptions that hinder the utility of machine learning approaches for malware recognition and more generic IDS domains. We then formalize an *adaptive open world framework* for stealth malware recognition, bringing together recent advances in several areas of machine learning literature including intrusion detection, novelty detection, and other recognition domains. Finally, Section VI concludes this survey.



(a) Proportion of Malware by Platform Type



(b) Growth Rate in Malware Proportion by Platform Type

Fig. 1. Malware proportions and rates by platform. As shown in (a), malware designed specifically for the Microsoft Windows family of operating systems accounts for over 90% of all malware. While malware for other platforms is growing rapidly, at current growth rates, shown in (b), the quantity of malware designed for any other platform is unlikely to surpass the quantity of Windows malware any time soon. Examining overall growth rates per platform, we see higher growth rates in non Windows malware, but a high growth rate on a small base is still quite small in terms of overall impact. For Windows the 88% growth rate is on a base of 135 million malware samples, which translates into about 118 million new Windows malwares. In comparison the high growth rate for Apple iOS, with an increase of more than 230%, is on a base of 30,400 samples, with the total number of discovered Apple malware samples in 2015 just under 70,000; very small compared to the number of Windows malware samples as well as the 4.5 milion Android malware samples. Numbers for these plots were obtained from a 2016 HP Enterprise threat report [13].

II. A Survey of Existing Stealth Malware

We discuss four types of stealth technology: rootkits, code mutation, anti-emulation, and targeting mechanisms. Before getting into the details of each, we summarize them at a

high level. Note that current malware usually uses a mixture of several or all concepts that are described in this section. For example, a rootkit might maintain malicious files on disk that survive reboots, while using hooking techniques to hide these files and processes so that they cannot be easily detected and removed, and applying code mutation techniques to prevent anti-malware systems from detecting running code.

Rootkit technology refers to software designed for two purposes: maintaining stealth presence and allowing continued access to a computer system. The stealth functionality includes hiding files, hiding directories, hiding processes, masking resource utilization, masking network connections, and hiding registry keys. Not all rootkit technology is malicious, for example, some anti-malware suites use their own rootkit technologies to evade detection by malware. Samhain [21], [22], for example, was one of the first pieces of anti-malware (specifically anti-rootkit) software to hide its own presence from the system, such that a malware or hacker would not be able to detect and, thus, kill off the Samhain process. Whether rootkit implementations are designed for malicious or benign applications, many of the underlying technologies are the same. In short, rootkits can be thought of as performing man-in-the-middle attacks between different components of the operating system. In doing so, different rootkit technologies employ radically different techniques. In this section, we review four different types of rootkits.

Unlike rootkit technologies, code mutation does not aim to change the dynamic functionality of the code. Instead, it aims to alter the appearance of code with each generation, generally at the binary level, so that copies of the code cannot be recognized by simple pattern-matching algorithms.

Due in part to the difficulties of static code analysis, and in part to protect system resources, the behavior of suspicious executables is often analyzed by running these executables in virtual sandboxed environments. *Anti-emulation* technologies aim to detect these sandboxes; if a sandbox is detected, they alter the execution flow of malicious code in order to stay hidden.

Finally, *targeting mechanisms* seek to manage the spread of malware and therefore minimize risk of detection and collateral damage, allowing it to remain in the wild for a longer period of time.

A. Type 1 Rootkits: Malicious System Files on Disk

Summary: Mimic system process files.
Advantages: Easy to install, survives reboots.
Disadvantages: Easy to detect and remove.

The first-generation of rootkits masqueraded as disk-resident system programs (e.g., `ls`, `top`) on early Unix machines, pre-dating the development of Windows. These early implementations were predominantly designed to obtain elevated privileges, hence the name "rootkit". Modern rootkit technologies are designed to maintain stealth, perform activity logging (e.g., key logging), and set up backdoors and covert channels for command and control (C2) server communication [3].

Although modern rootkits (types 2, 3, and 4) rely on privilege escalation for their functionalities, their main objective is stealth (although privilege escalation is often assumed) [23]. Since first-generation rootkits reside on disk, they are easily detectable via a comparison of their hashes or checksums to hashes or checksums of system files. Due to early file integrity checkers such as Tripwire [24], first-generation rootkits have greatly decreased in prevalence and modern rootkits have trended toward memory residency over disk residency [25], [26]. This should not be conflated with saying that modern malwares are trending away from disk presence – e.g., Gapz [27] and Olmasco [28] are advanced bootkits with persistent disk data. As we shall see below, many modern rootkits are specifically designed to intercept calls that enumerate files associated with a specific malware and hide these files from the file listing.

B. Type 2 Rootkits: Hooking and In-Memory Redirection of Code Execution

Summary: Code injection by modifying pointers to libraries/functions or by explicit insertion of code.
Advantages: Difficult to differentiate genuine and malicious hooking.
Disadvantages: Difficult to inject.

Second-generation rootkits hijack process memory and divert the flow of code execution so that malicious code gets executed. This rootkit technique is generally referred to as *hooking*, and can be done in several ways [23], e.g., via modification of function pointers to point to malicious code or via inline function patching – an approach involving overwriting of code, not just pointers [29]. For readability, however, we use the term hooking to refer to any in-memory redirection of code execution.

Rootkits use hooking to alter memory so that malicious code gets executed, usually prior to or after the execution of a legitimate operating system call [26], [29]. This allows the rootkit to filter return values or functionality requested from the operating system. There are three types of hooking [25]: user-mode hooking, kernel-mode hooking, and hybrid hooking.

Hooking in general is not an inherently malicious technique. Legitimate uses for hooking exist, including hot patching, monitoring, profiling, and debugging. Hooking is generally straightforward to detect, but distinguishing legitimate hooking instances from (malicious) rootkit hooking is a challenging task [25], [26].

1) User-Mode Hooking:

Summary: Injection of code into User DLLs.
Advantages: Difficult to classify as malicious.
Disadvantages: Easy to detect.

To improve resource utilization and to provide an organized interface to request kernel resources from user space, much of the Win32 API is implemented as dynamically linked libraries (DLLs) whose callable functions are accessible via tables of function pointers. DLLs allow multiple programs to share the same code in memory without requiring the code to be resident in each program's binary [30]. In and of themselves, DLLs are nothing more than special types of portable executable (PE) files. Each DLL contains

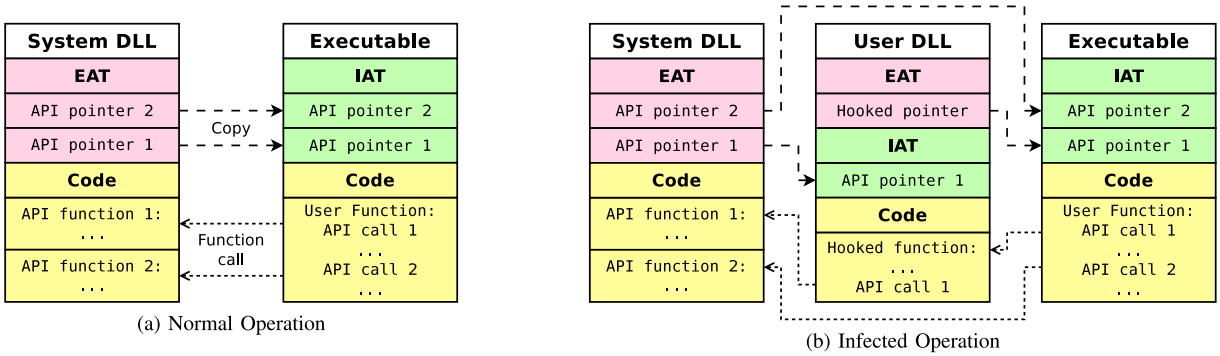


Fig. 2. Hooking. This figure shows an example of code redirection on shared library imports. (a) displays the normal operation of API calls, where API pointers of the DLL's EAT are copied into the executable's IAT. (b) shows how IAT hooking injects (malicious) code from a user DLL before executing the original API call 1.

an Export Address Table (EAT) of pointers to functions that can be called outside of the DLL. Other programs that wish to call these exported functions generally have an Import Address Table (IAT) containing pointers to the DLL functions in their PE images in memory. This lays the ground for the popular user-mode rootkit exploit known as *IAT hooking* [3], in which the rootkit changes the function pointers within the IAT to point to malicious code. Fig. 2 illustrates both malicious and legitimate usage of IAT hooks. In the context of rootkit IAT hooking, the functions hooked are almost always operating system API functions and the malicious code pointed to by the overwritten IAT entry, in addition to its malicious behavior, almost always makes a call to the original API function in order to spoof its functionality [3], [25]. Prior to or after the original API call, however, the malicious code causes the result of the library call to be changed or filtered. By interposing the FindFirstFile and FindNextFile Win32 API calls, for example, a rootkit can selectively filter files of specific unicode identifiers so that they will not be seen by the caller. This particular exploit might involve calling FindNextFile multiple times within the malicious code to skip over malicious files and protect its stealth.

IAT hooking is nontrivial and has its limitations [25], [30], [31], for example, it requires the PE header of the target binary to be parsed and the correct addresses of target functions to be identified. Practically, IAT hooking is restricted to OS API calls, unless specifically engineered for a particular non-API DLL [25], [31]. An additional difficulty of IAT hooking is that DLLs can be loaded at different times with respect to the executable launch [25]. DLLs can be loaded either at load time or at runtime of the executable. In the latter case, the IAT does not contain function pointers until just before they are used, so hooking the IAT is considerably more difficult. Further, by loading DLLs with the Win32 API calls LoadLibrary and GetProcAddress, no entries will be created in the IAT, making the loaded DLLs impervious to IAT hooking [25], [30].

Inline function patching, a.k.a. *detouring*, is another common second-generation technique, which avoids some of the shortcomings of IAT hooking [25]. Unlike function pointer modification, detouring uses the direct modification of code in memory. It involves overwriting a snippet of code with an unconditional jump to malicious code, saving the stub of code that was overwritten by the malicious code, executing the stub after the malicious code, and possibly jumping back to the point of departure from the original code so that the original code gets executed – a technique known as *trampolining* [25].

In practice, overwriting generic code segments is difficult for several reasons. First, stub-saving without corrupting memory is inherently difficult [26]. Second, the most common instruction sets, including x86 and x86-64, are variable-length instruction sets, meaning that disassembly is necessary to avoid splitting instructions in a generic solution [32]. Not only is disassembly a high overhead task for stealth software, but even with an effective disassembly strategy, performing arbitrary jumps to malicious code can result in unexpected behavior that can compromise the stealth of the rootkit [26]. Consider, for example, mistakenly placing a jump to shell code and back into a loop that executes for many iterations. One execution of the shell code might have negligible overhead, but a detour placed within an otherwise tight loop may have a noticeable effect on system behavior.

Almost all existing Windows rootkits that rely on inline function patching consequently hook in the first few bytes of the target function [25]. In addition to the fact that an immediate detour limits the potential for causing strange behaviors, many Windows compilers for x86 leave 5 NOP bytes at the beginning of each function. These bytes can easily be replaced by a single byte jump opcode and a 32 bit address. This is not an oversight. Rather, like hooking in general, detours are not inherently malicious and have a legitimate application, namely *hot patching* [33]. Hot patching is a technique, which uses detours to perform updates to binary in memory. During hot patching, an updated copy of the function is placed elsewhere in memory and a jump instruction with the address of the updated copy as an argument is placed at the beginning of the original function. The purpose of hot patching is to increase availability without the need for program suspension or reboot [34]. Microsoft Research even produced a software package called Detours specifically designed for hot patching [33], [34]. In addition, detours are also used in

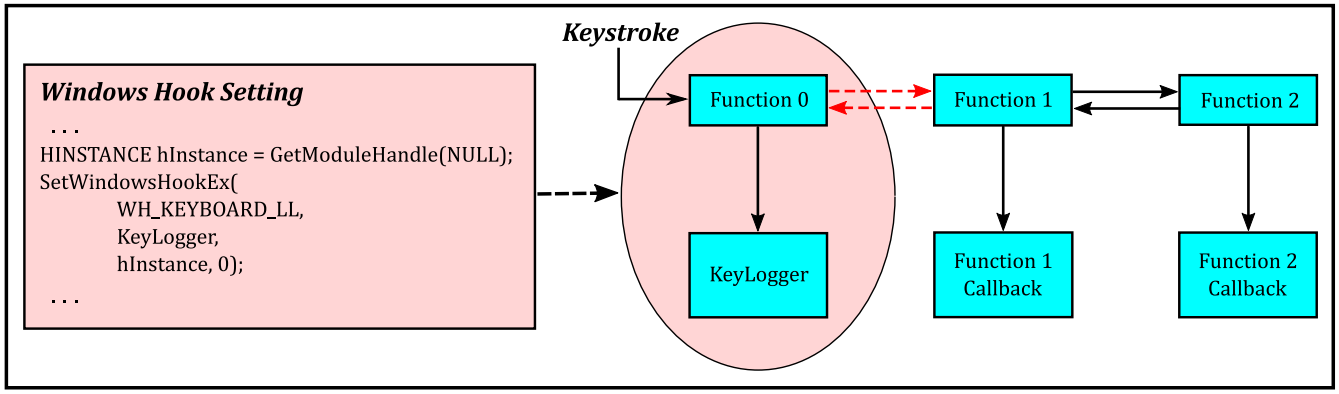


Fig. 3. Exploiting event hook chains. A prototypical keylogger application gets the target process' context, injects a malicious DLL into its address space, and prepends a function pointer to code within this DLL to the keypress event hook chain. Whenever a key is pressed, the newly introduced callback is triggered, thereby allowing the malicious code to log every keystroke.

anti-malware [25]. Like `IAT` hooks, detours are relatively easy to detect. However, the legitimate applications of detours are difficult to distinguish from rootkit uses of detours [25], [34].

Detours are also not limited in scope to user mode API functions. They can also be used to hook kernel functions [25]. Regardless of the hooking strategy, when working in user mode, a rootkit must place malicious code into the address space of the target process. This is usually orchestrated through *DLL injection*, i.e., by making a target process load a DLL into its address space. Having a process load a DLL is common, so common that DLL injection can simply be performed using Win32 API functionality. This makes DLL injections easy to detect. However, discerning benign DLL injections from malicious DLL injections is a more daunting task [35]–[37].

Three of the most common DLL injection techniques are detailed in [35]–[37]. The simplest technique exploits the AppInit_DLLs registry key, which proceeds as follows: a DLL containing a `DllMain` function is written, optionally with a payload to be executed. The DLL main function takes three arguments: a DLL handle, the reason for the call (process attach/detach or thread attach/detach), and a third value which depends on whether the DLL is statically or dynamically loaded. By changing the value of the AppInit_DLLs registry key to include the path to the DLL to be executed, and changing the `LoadAppInit_DLLs` registry key's value to 1, whenever any process loads `user32.dll`, the injected DLL will also be loaded and the `DllMain` functionality will be executed. Although the DLL gets injected only when a program loads `user32.dll`, `user32.dll` is prevalent in many applications, since it is responsible for key user interface functionality. Whether or not `DllMain` calls malicious functionality, the AppInit technique can be used to inject a DLL into an arbitrary process' address space, as long as that process calls functionality from `user32.dll`. Note that, although the injection itself involves setting a registry key, which could indicate the presence of a rootkit, the rootkit can change the value of the registry key once resident in the target process' address space [25].

A second method of DLL injection exploits Windows event hook chains [36], [38], [39]. Event hook chains are linked lists containing function pointers to application-defined callbacks. Different hook chains exist for different types of events, including key presses, mouse motions, mouse clicks, and messages [39]. Additional procedures can be inserted into hook chains using the `SetWindowsHookEx` Win32 API call. By default, inserted hook procedures are placed at the front of a hook chain, e.g., prominent rootkits/bootkits [28], [40] overwrite the pointer to the handlers in the `DRIVER_OBJECT` structure, but some rootkits, e.g., Win32/Gapz [27], use splicing, patching the handlers' code themselves. Upon an event associated with a particular hook chain, the operating system sequentially calls functions within the hook chain. Each hook function determines whether it is designed to handle the event. If not, it calls the `CallNextHookEx` API function. This invokes the next procedure within the hook chain. There are two specific types of hook chains: global and thread specific. Global hooks monitor events for all threads within the calling thread's desktop, whereas thread-specific hooks monitor events for individual threads. Global hook procedures must reside in a DLL disjoint from the calling thread's code, whereas local hook procedures may reside within the code of the calling thread or within a DLL [39].

For hook chain DLL injection a support program is required, as well as a DLL exporting the functionality to be hooked. The attack proceeds as follows [36]: the support program gets a handle to the DLL and obtains the address of one of the exported functions through Win32 API calls. The support program then calls the `SetWindowsHookEx` API function passing it the action to be hooked and the address of the exported function from the DLL. `SetWindowsHookEx` places the hook routine into the hook chain of the victim process, so that the DLL functionality is invoked whenever a specified event is triggered. When the event first occurs, the OS injects the specified DLL into the process' address space, which automatically causes the `DllMain` function to be called. Subsequent events do not require the DLL to be reloaded since the DLL's exports get called from within the victim process' address space. An example keylogger rootkit is shown in Fig. 3, which will log the pressed key and call `CallNextHookEx` to trigger the default handling of

the keystroke. Again, benign addition of hook chain functions via SetWindowsHookEx is common, e.g., to decide which window/process should get the keystroke; the difficult task is determining if any of the added functions are malicious.

A third common DLL injection strategy involves creating a remote thread inside the virtual address space of a target process using the CreateRemoteThread Win32 API call [37]. The injection proceeds as follows [37]: a support program controlled by the malware calls OpenProcess, which returns a handle to the target process. The support program then calls GetProcAddress for the API function LoadLibrary. LoadLibrary will be accessible from the target process because this API function is part of kernel32.dll, a user space DLL from which every Win32 user space process imports functionality. To insert the exported function name into the target process' address space, the malicious process must call the VirtualAllocEx API function. This API function allocates a virtual memory range within the target process' address space. The allocation is required in order to store the name of the rootkit DLL function. The WriteProcessMemory API call is then used to place the name of the malicious DLL into the target process' address space. Finally, the CreateRemoteThread API function calls the LoadLibrary function to inject the rootkit DLL. Like event chains, the CreateRemoteThread API call has legitimate uses. For example, a debugger might fire off a remote thread in a target process' address space for profiling and state inspection. An anti-malware module might perform similar behavior. Finally, IO might be handled through pointers to callbacks exchanged by several processes, where the callback method is intended to execute in another process' address space. The fact that the API call has so many potentially legitimate uses makes malicious exploits particularly difficult to detect.

2) Kernel-Mode Hooking:

Summary: Injection of code into the Kernel via device drivers.

Advantages: Difficult to detect by user-mode IDSs.

Disadvantages: Intricate to implement correctly.

Rootkits implementing kernel hooks are more difficult to detect than those implementing user space hooks. In addition to the extended functionality afforded to the rootkit, user space anti-malwares cannot detect kernel hooks because they do not have the requisite permissions to access kernel memory [25], [26]. Kernel memory resides in the top part of a process' address space. For 32-bit Windows, this usually corresponds to addresses above 0x80000000, but can correspond to addresses above 0xC0000000, if processes are configured for 3GB rather than 2GB of user space memory allocation. All kernel memory across all processes maps to the same physical location, and without special permissions, processes cannot directly access this memory.

Kernel hooks are most commonly implemented as device drivers [25]. Popular places to hook into the kernel include the System Service Descriptor Table (SSDT), the Interrupt Descriptor Table (IDT), and I/O Request Packet (IRP) function tables of device drivers [25]. The SSDT was the hooking mechanism used in the classic Sony DRM Rootkit [41] and the more recent Necurs malware [42], and is often a used as part of more complex multi-exploitation kits such as the RTF zero-day (CVE-2014-1761) attack [43] which was detected in the wild.

The SSDT is a Windows kernel memory data structure of function pointers to system calls. Upon a system call, the operating system indexes into the table by the function call ID number, left-shifted 2 bits to find the appropriate function in memory. The System Service Parameter Table (SSPT) stores the number of bytes that arguments require for each system call. Since SSPT entries are one byte each, system calls can take up to 255 arguments. The KeServiceDescriptorTable contains pointers to both the SSDT and the SSPT.

When a user space program performs a system call, it invokes functionality within ntdll.dll, which is the main interface library between user space and kernel space. The EAX register is filled with the system function call ID and the EDX register is filled with the memory address of the arguments. After performing a range check, the value of EAX is used by the OS to index into the SSDT. The program counter register IP is then filled with the appropriate address from the SSDT, executing the dispatcher call. Dispatches are triggered by the SYSENTER processor instruction or the more dated INT 2E interrupt.

SSDT hooks are particularly dangerous because they can supplement any system call with their own functionality. Hoglund and Butler [25] provide an example of process hiding via SSDT hook, in which the NTQuerySystemInformation NTOS system call is hooked to point to shell code, which filters ZwQuerySystemInformation structures corresponding to processes by their unicode string identifiers. Selective processes can be hidden by changing pointers in this data structure. Windows provides some protection to prevent SSDT hooks by making SSDT memory read-only. Although this protection makes the attacker's job more difficult, there are ways around it. One method is to change the memory descriptor list (MDL) [25], [44] for the requisite area in memory. This involves casting the KeServiceDescriptorTable to the appropriate data structure, and using it to build an MDL from non-paged memory. By locking the memory pages and changing the flags on the MDL one can change the permissions on memory pages. Another method of disabling memory protections is by zeroing the write protection bit in control register CR0.

The Interrupt Descriptor Table (IDT) is another popular hook target [45]. The interrupt table contains pointers to callbacks that occur upon an interrupt. Interrupts can be triggered by both hardware and software. Because interrupts have no return values, IDT hooks are limited in functionality to denying interrupt requests. They cannot perform data filtration. Multiprocessing systems have made IDT hooking more difficult [25]. Since each CPU has its own IDT, an attacker must usually hook IDTs of all CPUs to be successful. Hooking only one

of multiple IDTs causes an attack to have only limited impact.

A final popular kernel hook target discussed by Hoglund and Butler [25] is the IRP dispatch table of a device driver. Since many devices access kernel memory directly, Windows abstracts devices as device objects. Device objects may represent either physical hardware devices such as buses or they may represent software "devices" such as network protocol stacks or updates to the Windows kernel. Device objects may even correspond to anti-virus components that monitor the kernel. Each device object has at least one device driver. Communication between kernel and device driver is performed via the I/O manager. The I/O manager calls the driver, passing pointers to the device object and the I/O request. The I/O request is passed in a standardized data structure called an I/O Request Packet (IRP). Within the device object is a pointer to the driver object. Drivers themselves are nothing more than special types of DLLs. The I/O manager passes the device object and the IRP to the driver. How the driver behaves depends on the contents and flags of the IRP. The function pointers for various IRP options are stored in a dispatch table within the driver. A rootkit can subvert the kernel by changing these function pointers to point to shell code [25]. An anti-virus implemented as a filter driver, for example, may be subverted by rewriting its dispatch table. Hoglund and Butler [25] provide an in-depth example of using driver function table hooking to hide TCP connections. Essentially any kernel service that uses a device driver can be subverted by hooking the IRP dispatch table in a similar manner.

Note that while hooking device driver dispatch tables sounds relatively simple, the technique requires sophistication to be implemented correctly [25]. First, implementing bug-free kernel driver code is an involved task to begin with. Since drivers share the same memory address space as the kernel, a small implementation error can corrupt kernel memory and result in a kernel crash. This is one of the reasons that Microsoft has gravitated to user-mode drivers when possible [46]. Second, in many applications drivers are stacked. When dealing with physical devices, the lowest level driver on the stack serves the purpose of abstracting bus-specific behavior to an intermediate interface for the upper level driver. Even in software, drivers may be stacked, for example, anti-virus I/O filtering or file system encryption/decryption can be performed by a filter driver residing in the mid-level of the stack [25]. A successful rootkit author must therefore understand how the device stack behaves, where in the device stack to hook, and how to perform I/O completion such that the hook does not result in noticeably different behavior.

3) Hybrid Hooking:

Summary: Hook user functions into kernel DLLs.

Advantages: Even more difficult to detect than kernel hooking.

Disadvantages: More difficult to implement than kernel hooking.

Hybrid hooks aim to circumvent anti-malwares by attacking user space and kernel space simultaneously. They involve implementing a user space hook to kernel memory. Hoglund and Butler [25] discuss a technique to hook the user space IAT of a process from the kernel. The motivation behind this technique is based on the observation that user space IAT hooks are detectable because one needs to allocate memory within the process' context or inject a DLL for the same effect. But is there some means to hook the IAT through the kernel, without the need to allocate user space memory for IAT hooks? The answer is yes: the attack in [25] leverages two aspects of the Windows architecture. First, it uses the PsSetLoadImageNotifyRoutine, a kernel mode support routine that registers driver callback functions to be called whenever a PE image gets loaded into memory [47]. The callback is called within the target process' context after loading but before execution of the PE. By parsing the PE image in memory, an attacker can change the IAT. The question then becomes, how to run malicious code without overt memory allocation or DLL injection into the process' address space? One solution uses a specific page of memory [48]: in Windows there exists a physical memory address shared by both kernel space and user space, which the kernel can write to, but the user cannot. The user and kernel mode addresses are 0x7FFE0000 and 0xFFDF0000, respectively. The reason for this virtual $\leftrightarrow$ physical mapping convention stems from the introduction of SYSENTER and SYSEXIT processor instructions, for fast switches between user mode and kernel mode. Approximately 1kB of this page is used by the kernel [25], but the remaining 3kB are blank. Writing malicious code to addresses in the page starting at 0xFFDF0000 in kernel space and placing a function pointer to the beginning of the code at the corresponding address in user space allows the rootkit to hook the IAT without allocating memory or performing DLL injection.

Another hybrid attack is discussed in [49]. The attack is called Illusion, and involves both kernel space and user space components. The motivation behind the attack is to circumvent intrusion detection systems that rely on system call analysis (see Section III). To understand the Illusion attack, we review steps involved in performing a system call: first, a user space application issues an INT 3 interrupt or a SYSENTER processor instruction, which causes the processor to switch to kernel mode and execute the dispatcher. The dispatcher indexes into the SSDT to find the handler for the system call in question. The handler performs its functionality and returns to the dispatcher. The dispatcher then passes return values to the user space application and returns the processor to user space. These steps should be familiar from the prior discussion of hooking the SSDT. Illusion works by creating a one-to-all mapping of potential execution paths between system calls, which take array buffer arguments and the function pointers of the SSDT. Although the same effect could be obtained by making changes directly to the SSDT, the Illusion approach, unlike SSDT hooking, cannot be detected using the techniques discussed in Section III. Illusion exploits system calls such as DeviceIoControl, which is used to exchange data buffers between application and kernel driver. Parts of

the rootkit reside in both in kernel space and in user space. Messages are passed between user space rootkit and kernel space rootkit by filling the buffer. Communication is managed via a dedicated protocol. This allows the user space rootkit to make system calls on its behalf without changing the SSDT. Further, metamorphic code as described in Section II-E can be leveraged to change the communication protocol at each execution.

C. Type 3 Rootkits: Direct Kernel Object Manipulation

Summary: Modify dynamic kernel data structures.
Advantages: Extremely difficult to detect.
Disadvantages: Has limited applications.

Although second-generation rootkits remain ubiquitous, they are not without their limitations. Their change of overt function behavior inherently leaves a detectable footprint because it introduces malicious code – either in user space or in kernel space – which can be detected and analyzed [23]. Third-generation *direct kernel object manipulation* (DKOM) attacks take a different approach. DKOM aims to subvert the integrity of the kernel by targeting dynamic kernel data structures responsible for bookkeeping operations [23]. Like kernel space hooks, DKOM attacks are immune to user space anti-malware, which assumes a trusted kernel. DKOM attacks are also much harder to detect than kernel hooks because they target dynamic data structures whose values change during normal runtime operation. By contrast, hooking static areas of the kernel like the SSDT can be detected with relative ease because these areas should remain constant during normal operation [23].

The canonical example of DKOM is process hiding. The attack can be carried out on most operating systems, and relies on the fact that schedulers use different data structures to track processes than the data structures used for resource bookkeeping operations [50]. In the Windows NTOS kernel, for example, the kernel layer[1] is responsible for managing thread scheduling, whereas the executive layer, which contains the memory manager, the object manager, and the I/O manager is responsible for resource management [46]. Since the executive layer allocates resources (e.g., memory) on a per-process basis, it views processes as EPROCESS (executive process) data structures, maintained in double circularly linked lists. The scheduler, however, operates on a per-thread instance, and consequently maintains threads in its own double circularly linked list of KTHREAD (kernel thread) data structures. By modifying pointers, a rootkit with control over kernel memory can decouple an EPROCESS node from the linked list, re-coupling the next and previous EPROCESS structures' pointers. Consequently, the process will no longer be visible to the executive layer and calls by the Win32 API will, therefore, not display the process. However, the thread scheduler will continue CPU quantum allocation to the threads corresponding to the hidden EPROCESS node. The process will, thus, be effectively invisible to both user and kernel mode programs – yet it will still continue to run. This attack is depicted in Fig. 4.

[1]The kernel itself has three layers, one of which is called the *kernel layer*. The other two layers of the kernel are the *executive layer* and the *hardware abstraction layer*.

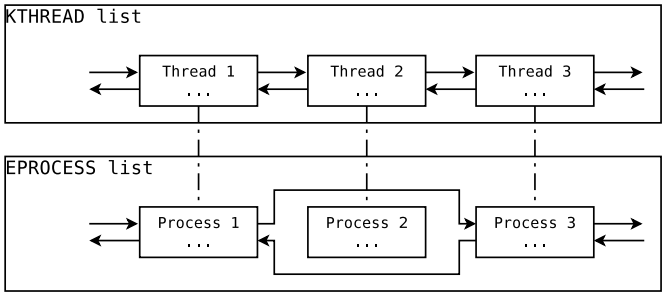


Fig. 4. DKOM attack. This figure displays a successful DKOM attack, where the malicious code of Process 2 is hidden from the system yet continues to execute.

While process hiding is the canonical DKOM example, it is just one of several DKOM attack possibilities. Baliga *et al*. [51] discuss several known DKOM attack variants, including zeroing entropy pools for pseudorandom number generator seeds, disabling pseudorandom number generators, resource waste and intrinsic DOS, adding new binary formats, disabling firewalls, and spoofing in-memory signature scans by providing a false view of memory.

Proper DKOM implementations are extremely difficult to detect. Fortunately, DKOM is not without its shortcomings and difficulties from the rootkit developer's perspective. Changing OS kernel data structures is no easy task and incorrect implementations can easily result in kernel crashes, thereby causing an overt indication of a malware's presence. Also, DKOM introduces no new code to the kernel apart from the code to modify kernel data structures to begin with. Therefore, inherent limitations on the scope of a DKOM attack are imposed by the manner in which the kernel uses its data structures. For example, one usually cannot hide disk resident files via DKOM because most modern operating systems do not have kernel level data structures corresponding to lists of files.

D. Type 4 Rootkits: Cross Platform Rootkits and Rootkits in Hardware

Summary: Attack systems using very low-level rootkits.
Advantages: Undetectable by conventional software countermeasures.
Disadvantages: Requires custom low-level hypervisor, BIOS, hardware or physical/supply chain compromise to be effective.

Fourth-generation rootkit technologies operate at the virtualization layer, in the BIOS, and in hardware [52]. To our knowledge, fourth-generation rootkits have been developed only in proof-of-concept settings, as we could not find any documentation of fourth-generation rootkits in the wild. Because they reside at a lower level than the operating system, they cannot be detected through the operating system and are, therefore, OS independent. However, they still dependent on the type of BIOS version, instruction set, and hardware [52]. Since fourth-generation rootkits are theoretical in nature – at least as of now – we consider them outside the scope of this survey. We mention them in this section for completeness and because they may become relevant after the publication of this survey.

E. Code Mutation

Summary: Self-modifying malicious code.
Advantages: Avoids simple signature matching.
Disadvantages: Greater runtime overhead and detectable via emulation.

In early viruses, the viral code was often appended to the end of an executable file, with the entry point changed to jump to the viral code before running the original executable [26]. Once executed, the virus code in turn would jump to the beginning of the body of the executable so that the executable was run post-replication. The user would be none the wiser until the host system had been thoroughly infected. Anti-malware companies soon got wise and started checking hashes of code blocks – generally at the end of files. To counter, malware authors began to encrypt the text of the viruses. This required a decryption routine to be called at the beginning of execution. The virus was then re-encrypted with a different key upon each replication [26].

These encrypted viruses had a fatal flaw: the decryption routine was jumped to somewhere in the executable. Anti-malware solutions merely had to look for the decrypter. Thus, *polymorphic* engines were created, in which the decryption engine mutated itself at each generation, no longer matching a fixed signature. However, polymorphic viruses were still susceptible to detection [26]: although the detector mutated, the size of the malicious code did not change, was still placed at the end of the file, and was susceptible to entropy analysis, depending on the encryption technique.

To this end, entry-point obscuring (EPO) viruses were created, where the body of the viral code is placed arbitrarily in the executable, sometimes in a distributed fashion [26]. To counter the threats from polymorphic viruses, Kaspersky (of Kaspersky Lab fame) and others [54] created emulation engines, which run potentially malicious code in a virtual machine. In order to run, the body of the viral code must decrypt itself in memory in some form or another, and when it does, the body of the malicious code is laid bare for hashed signature comparison as well as behavioral/heuristic analysis.

To combat emulation, *metamorphic* engines were developed. Just as polymorphic malwares mutate their decryption engines at each generation, metamorphic engines mutate the full body of their code and, unlike polymorphics, change the size of the code body from one generation to another [26]. Some malwares still encrypt metamorphic code, or parts of metamorphic code, while others do not – as encryption and run time packing techniques can reveal the existence of malicious code [26].

Metamorphic code mutation techniques, as shown in Fig. 5, include register swaps, subroutine permutations, transpositions of independent instructions, insertion of NOPs or instruction sequences that behave as NOPs, and parser-like mutations by context-free grammars (or other grammar types) [54]–[57]. Many metamorphic techniques are similar to compilation techniques, but for a much different purpose. The metamorphic engine in the MetaPHOR worm, for example, disassembles executable code into its own intermediate representation and uses its own formal grammar to carry out this mutation [54].

Code transformation techniques are not particular to native code either: Faruki *et al*. [58] presented several Dalvik bytecode obfuscation techniques and tested them against several Android security suites, which often failed at recognizing transformed malicious code. While some of the transformation targets are unique to obfuscation on Android devices – e.g., renaming packages and encrypting resource files – the control, data, and layout transformations in [58] follow the same principles of code obfuscation at the native level.

F. Anti-Emulation

Summary: Malware behaves differently when running in an emulated environment.
Advantages: Malware evades detection during emulation.
Disadvantages: Needs to detect the presence of the emulator reliably. May not run in certain virtualized environments.

Mutation engines, including metamorphics and polymorphics, change the instructions in the target code itself and naturally its runtime [53]. However, they do not change the underlying functionality. Therefore, during emulation (see Section III-E), behavioral and heuristic techniques can be used to fingerprint malicious code, for example, if the malware conducts a strange series of system calls, or if it attempts to establish a connection with a C2 server at a known malicious address. Hence, malware can be spotted regardless of the degree of obfuscation present in the code [26].

The success of early emulation techniques led to the usage of malicious anti-emulation tactics, which include attempts to detect the emulator by examining machine configurations – e.g., volume identifiers and network interface – and use of difficult to emulate functionality, e.g., invoking the GPU [26], [59]. In turn, emulation strategies have become more advanced, for example, in their DroidAnalyst framework [59] for Android, Faruki *et al*. implement a realistic emulation platform by overloading default serial numbers, phone numbers, geolocations, system time, email accounts, and multimedia files to make their emulator more difficult to detect.

A realistic emulation environment is a good start to avoid emulator detection based on hardware characteristics, but it alone is insufficient to defeat all types of anti-emulation, for example, Duqu only executes certain components after 10 minutes idle when certain requirements are met [60]. Similarly the Kelihos botnet [61], and the Nap Trojan [62] use the `SleepEx` and `NtDelayExecution` API calls to delay malicious execution until longer times than a typical emulator will devote to analysis. PoisonIvy [63] and similarly UpClicker establish malicious connections only when the left mouse button is released. PushDo takes a more offensive approach, using `PspCreateProcessNotify` to de-register sandbox monitoring routines [65]. Other malwares take advantage of dialog boxes and scrolling [65]. Even mouse movements are taken into consideration and malware can differentiate between human and simulated mouse movements by assessing speed, curvature, and other features [65]. Thus, emulated environments for stealth malware detection face the

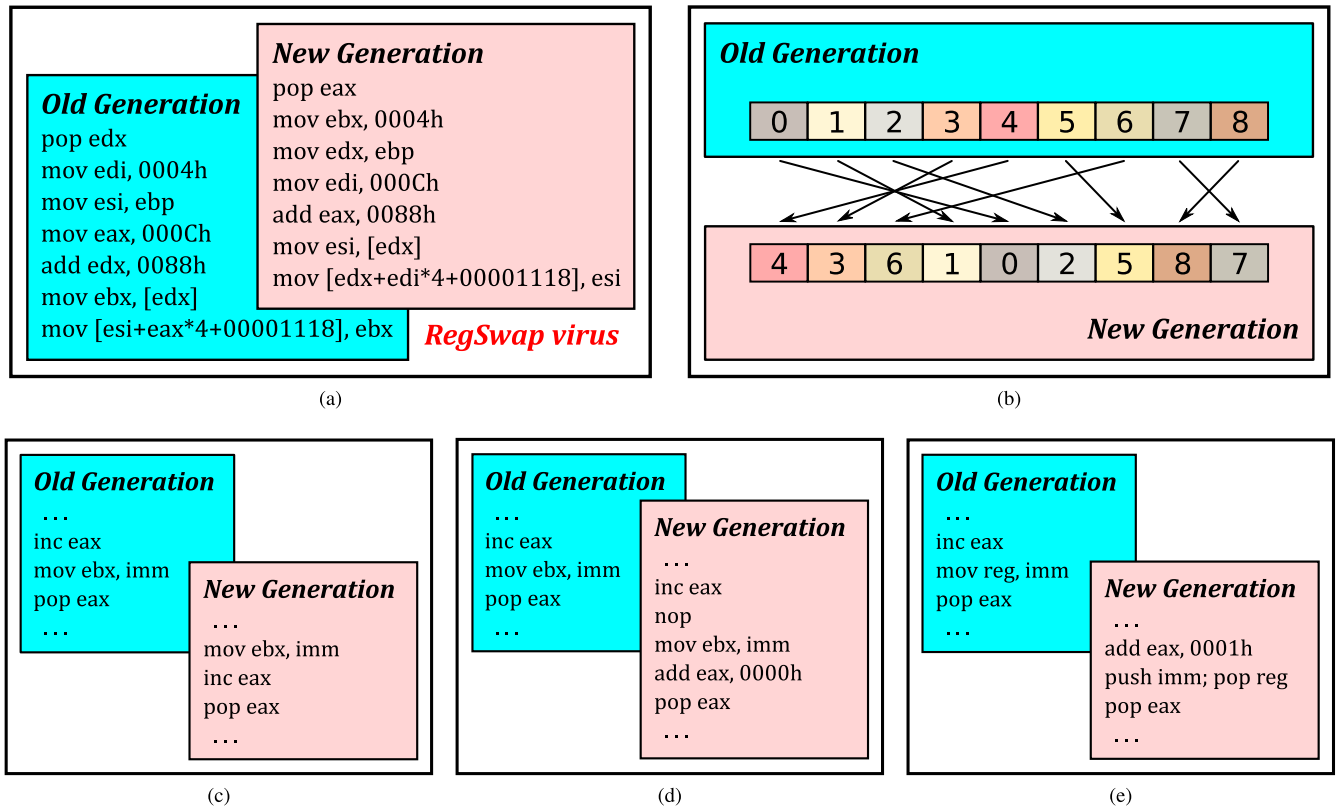


Fig. 5. Metamorphic code obfuscation. Five techniques employed by metamorphic engines to evade signature scans across malware generations. (a) Register swap: exchanging registers as demonstrated by code fragments from the RegSwap virus [53]. (b) Subroutine permutation: reordering subroutines of the virus code. (c) Transposition: modifying the execution order of independent instructions. (d) Semantic NOP-insertion: injecting NOPs or instructions that are semantically identical to NOPs. (e) Code mutation: replacing instructions with semantically equivalent code.

tradeoff between realistic emulation and implementation cost. Anti-emulation in turn faces a different problem: with the explosion of virtualization technology, thanks largely to the heavy drive toward cloud computing, virtualized (emulated) environments are seeing increased general-purpose use. This draws into question the effectiveness of anti-emulation as a stealth technique: if malicious code will not run in a virtual environment, then it might not be an effective attack if the targeted machine is virtualized.

G. Targeting Mechanisms

Summary: Malware runs on or spreads only to chosen systems.

Advantages: Decreases risk of detection.

Disadvantages: Malware spreads at a lower rate. Motivation for the attack is given away if detected.

Stealth targeted attacks – which aim to compromise specific targets – are becoming more advanced and more widespread [66]. While targeting mechanisms are not necessarily designed for stealth purposes, they have the effect of prolonging the amount of time that malware can remain undetected in the wild. This is done by allowing the malware to spread/execute only on certain high-value systems, thus minimizing the likelihood of detection while maximizing the impact of the attack. For example, recent point of sale (POS) compromises [67] targeted only specific corporations. The DarkHotel [68] *advanced persistent threat* (APT) targets only certain individuals (e.g., business executives). The notorious Stuxnet worm and its relatives Duqu, Flame, and Gauss employed sophisticated targeting mechanisms [60], preventing the malwares from executing on un-targeted systems. Stuxnet checks system configuration prior to execution; malicious components simply will not execute if the detected environment is not correct rather attempting to execute and failing [69]. Gauss's Gödel module is even encrypted with an RC4 cipher, with a key derived from system-specific data; thus, much of the functionality of the malware remains unknown, since a large part of the body of the code can only be decrypted with knowledge of the targeted machines [70]. Hence, IDS developers and anti-malware researchers cannot get the malicious code running in un-targeted machines. Targeting mechanisms may also change the behavior of the malware depending on the configuration of the machine so as to evade detection. For example, Flame dynamically changes file extensions depending on the type of anti-malware that it detects on the machine [71]. Other malwares may simply not run or choose to uninstall themselves to evade detection, while others will execute only under certain conditions on time, date, and geolocation [26], [60].

III. Component-Based Stealth Malware Countermeasures

In this section, we discuss anti-stealth malware techniques that aim to protect the integrity of areas of systems, which are

known to be vulnerable to attacks. These techniques include hook detection, cross-view detection, invariant specification, and hardware and virtualization solutions.

When assessing the effectiveness of any malware recognition system, it is important to consider the system's respective precision-recall tradeoff. Recall refers to the proportion of malicious samples of a given type that were correctly detected as malicious samples of that type, while precision refers to the proportion of the samples that the system marked as a malicious type that are actually of that malicious type. Increased recall tends to decrease precision, whereas increased precision tends to decrease recall. The "optimal" tradeoff between precision and recall for a given system depends on the application at hand. The integrity based solutions discussed in this section tend to offer higher precision rates than the pattern recognition techniques discussed in Section IV, but they are difficult to update because custom changes to hardware and software are required, making scalability an issue.

It is important to realize that the *component protection* techniques presented in this section are in practice often combined with more generic pattern recognition techniques discussed in Section IV [1], [26], [65], for example, hardware and virtualization solutions might be used to achieve a clean view of memory, on which a signature scan can be run [22], [72].

A. Detecting Hooks

Summary: Detect malwares that use hooking.
Advantages: Easy to implement.
Disadvantages: High false positive rates from legitimate benign hooks.

If a stealth malware uses in-memory hooks as described in Section II-B, IDSs can detect the malware by detecting its hooks. Unfortunately, methods that simply detect hooks trigger high false alarm rates since hooks are not inherently malicious. This makes weeding out false positives a challenging task. Also, since DKOM is not a form of hooking, hook detection techniques cannot detect DKOM attacks.

Ironically, an effective approach to detect hooks is to hook common attack points. By doing so, an anti-malware may not only be able to detect a rootkit loading into memory, but may also be able to preempt the attack. This might be accomplished by hooking the API functions used to inject DLLs into a target process' context (see Section II-B1) [25]. However, one must know what functions to hook and where to look for malicious attacks. Pinpointing attack vectors is not easy. For example, symbolic links are often not resolved to a common name until system call hooks have been executed [25]. Therefore, if the anti-malware relies on hooking the `SSDT` alone and matching the name of the target in the hook routine, an attacker can simply use an alias. Once hooks are observed, some tradeoff between precision and recall must be made: One can easily catch all rootkits loading into memory, and in doing so, create a completely unusable system (i.e., very high recall rates but extremely low precision rates).

Hook detection can be combined with signature and heuristic scans (discussed in Section IV) for ingress point monitoring. Based on a signature of the hooked code, the ingress point monitoring system can determine whether or not to raise an alarm. In contrast to trying to detect rootkit hooks as a rootkit loads, VICE [25], [29] uses memory scanning techniques that periodically inspect likely target locations of hooks such as the `IAT`, `SSDT`, or `IDT`. VICE detects hooks based on the presence of unconditional jumps to memory values outside of acceptable address ranges. Acceptable ranges can be defined by `IAT` module ranges, driver address ranges, and kernel process address ranges. For example, a system call in the `SSDT` should not point to an address outside `ntoskrnl.exe`.

Generic inline hooks cannot feasibly be detected via this method. Fortunately, as we discussed in Section II-B, hooks beyond the first few bytes of a function are rare, since they can result in strange behaviors, including noticeable slow down and outright program failure. For `SSDT` functions, unconditional jumps within the first few bytes outside of `ntoskrnl.exe` are indicators of hooks. `IAT` range checks require context switching into the process in question, enumerating the address ranges of the loaded DLLs, checking whether the function pointers in the `IAT` fall outside of their corresponding ranges, and recursively repeating this for all loaded DLLs.

A similar approach to VICE was taken in the implementation of System Virginity Verifier [73], which attempts to separate malicious hooking from benign hooking by comparing the in-memory code sections of drivers and DLLs to their disk images. Since these sections are supposed to be read-only, they should match in most cases, with the exception of a few lines of self-modifying kernel code in the NTOS kernel and hardware abstraction layers. Malicious hooks distinguish themselves from benign hooks when they exhibit discrepancies between in-memory and on-disk PE images, which will not occur under benign hooking [73]. Additionally, if the disk image is hidden then the hook likely corresponds to a rootkit. One must be careful in this case to distinguish missing files, which can occur in legitimate hooking applications, from hidden files. Other examples of image discrepancies associated with malicious hooks include failure of module attribution and code obfuscation.

An indirect approach to detecting hooks was implemented in Patchfinder 2 [74], in the form of API call tracing. This approach counts the number of instructions executed during an API call and compares the count to the number of instructions executed in a clean state. The intuition is based on the observation that in the context of rootkits, hooks almost always add instructions [74]. The technique requires proper baselining, which presents two challenges: first, deducing that the system is in a non-hooked state to begin with is difficult to establish, unless the system is fresh out of the box. Second, the Win32 API has many functions, which take many different arguments. Since enumerating all argument combination possibilities while acquiring the baseline is infeasible, API calls can vary substantially in instruction count even when unhooked.

B. Cross-View Detection and Specification Based Methods

Summary: Compare the output of API calls with that of low-level calls that are designed to do the same thing.
Advantages: Detects malware that hijacks API calls.

Disadvantages: Requires meticulous low-level code for to replicate functionality of most of the system API.

Cross-view detection is a technique aimed to reveal the presence of rootkits. The idea behind cross-view detection [75] is to observe the same aspect of a system in multiple ways, analogous to interviewing witnesses at a crime scene: just as conflicting stories from multiple witnesses likely indicate the presence of a liar, if different observations of a system return different results, the presence of a rootkit is likely. First, OS objects – processes, files, etc. – are enumerated via system API calls. This count is compared to that obtained using a different approach not reliant on the system API. For example, when traversing the file system, if the results returned by `FindFirstFile` and `FindNextFile` are inconsistent with direct queries to the disk controller, then a rootkit that hides files from the system is likely present.

One of the advantages of cross-view detection is that – if implemented correctly – maliciously hooked API calls can be detected with very few false positives because legitimate applications of API hooking rarely change the outputs of the API calls. Depending on the implementation, cross-view detection may or may not assume an intact kernel, and therefore may even be applied to detect DKOM. The main disadvantage of cross-view detection is that it is difficult to implement, especially for a commercial OS [76]. API calls are provided for a reason: to simplify the interface to kernel and hardware resources. Cross-view detection must circumvent the API, in many cases providing its own implementation. Theoretically, in most cases combinations of other API calls could be used in place of a from-scratch implementation. However, API call combinations are susceptible to the risk of other hooked API calls or duplicate calls to the same underlying code for multiple API functions, a common feature of the Win32 API [46].

Several cross-view detection tools have been developed over the years. Rootkitrevealer [77] by Windows SysInternals applies a cross-view detection strategy for the purposes of detecting persistent rootkits, i.e., disk-resident rootkits that survive across reboots. Rootkitrevealer uses the Windows API to scan the file system and registry, and compares the results to a manual parsing of the file system volume and registry hive. Klister [74] detects hidden processes in Windows 2000 by finding contradictions between executive process entries and kernel process entries used by the scheduler. Blacklight [78] combines both hidden file detection and hidden process detection. Microsoft's Strider Ghostbuster [50] is similar to Rootkitrevealer, except that it also detects hidden processes and it has the ability to compare an "inside the box" infected scan with an "outside the box" scan, in which the operating system is booted from a clean version.

If properly applied, cross-view detection offers high precision rootkit detection [76]. However, cross-view detection alone provides little insight on the *type of the rootkit* and must be combined with recognition methods (e.g., signature/behavioral) to attain this information [26], [76]. Cross-view detection methods are also cumbersome to update because they require new code, often interfacing with the kernel. Determining, which areas to cross-view, is also a challenging task [76].

C. Invariant Specification

Summary: Define constraints of an uninfected system.
Advantages: Detects DKOM attacks reliably.
Disadvantages: Constraints need to be well-specified, often by hand, and are highly platform-dependent.

A related approach to cross-view detection, especially applied to detecting DKOM, involves pinpointing kernel invariants – aspects of the kernel that should not change under normal OS behavior – and periodically monitoring these invariants. One example of a kernel invariant is that the length of the executive and kernel process linked lists should be equal, which is violated in the case of process hiding (see Fig. 4). Petroni *et al.* [79] introduce a framework for writing security specifications for dynamic kernel data structures. Their framework consists of five components: a low-level monitor used to access kernel memory, a model builder to synthesize the raw kernel memory binary into objects defined by the specification, a constraint verifier that checks the objects constructed by the model builder against the security specifications, response mechanisms that define the actions to take upon violation of a constraint, and finally, a specification compiler, which compiles specification constraints written in a high-level language into a form readily understood by the model builder.

Compelling arguments can be made in favor of the kernel-invariant based security specification approaches described above [79]: first, they allow a decoupling of site-specific constraints from system-specific constraints. An organization may have a security policy that forbids behavior not in direct violation of proper kernel function (e.g., no shell processes running with root UUID in Linux). Via a layered framework, specifications can be added without changing low-level implementations. Unlike signature-based approaches relying on rootkits having overlapping code fragments with other malwares, kernel-invariant specifications catch all DKOM attacks that violate particular specification constraints with only few false positives. The specification approach can even be extended beyond DKOM. However, using kernel invariant specification is not without its own difficulties. Proper and correct framework implementation is a tremendous programming effort in itself [79]. For closed-source operating systems like Windows, full information about kernel data structures and their implementation is seldom available, unless the specification framework tool is being developed as part of or in cooperation with the operating system vendor. Specification approaches can also exhibit false positives, for example, if kernel memory is accessed asynchronously via an external PCI interface like Copilot [22], a legitimate kernel update to a data structure may trigger a false positive detection simply because the update has not completed. Finally, the degree to which the invariant-specification approach works depends on the quality of the specification [79]. Correct specifications require in-depth domain specific knowledge about the kernel and/or about the organization's security policy. Due to the massive sizes and heterogeneities of operating systems,

even those of similar distribution, discerning a complete list of specifications is implausible without incorrect specifications that result in false positives. While a similar approach may have applications to other types of stealth malwares, Petroni *et al*. [79] introduced invariant specification as specific solution tailored to DKOM rootkits. Although invariant specification provides more readily available diagnostic information than cross-view detection because it tells which invariants are violated, invariant specification cannot discern the type of DKOM rootkit. Hence, more generic signature/behavioral techniques are required.

D. Hardware Solutions

Summary: Via hardware interface, use a clean machine to monitor another machine for the presence of rootkits or stealth malware

Advantages: Does not require an intact kernel on the monitored machine.

Disadvantages: Cannot interpose execution of malicious code.

The key motivation behind hardware based integrity checking is quite simple: a well-designed rootkit that has successfully subverted the OS kernel, or theoretically even the virtual layer and BIOS of a host machine, can return a spurious view of memory to a host based intrusion detection system (HIDS) such that the HIDS has no way of detecting the attack because its correct operation requires an intact kernel. Rather than relying on the kernel to provide a correct view of kernel memory, hardware solutions have been developed. For example, Copilot [22] uses direct memory access (DMA) via the PCI bus to access kernel memory from the hardware of the host machine itself and displays that view of memory to another machine. This in turn subverts any rootkit's ability to change the view of memory, barring a rootkit implemented in hardware itself. Depending on the hardware integrity checker in question, further analysis of kernel memory on the host machine may be performed via a supervisory machine alone, or alternatively with the aid of additional hardware. Copilot uses a coprocessor to perform fast hashes over static kernel memory and reports violations to a supervisory machine. Analysis mechanisms similar to those in [49] and [80] are employed on the supervisory machine in conjunction with DMA in order to properly parse kernel memory.

Using DMA to observe the memory layout of the host system from a supervisory system is appealing since a correct view of host memory is practically guaranteed. However, like all of the techniques that we have discussed, hardware based integrity checking is no silver bullet. In addition to the added expense and annoyance of requiring a supervisory machine, DMA based rootkit detection techniques can only detect rootkits, but they cannot intervene in the hosts execution. They have no access to the CPU and, therefore, cannot prevent or respond to attacks directly. This CPU access limitation not only means that CPU registers are invisible to DMA, it also means that the contents of the CPU cache cannot be inspected, leaving the theoretical possibility of a rootkit hiding malicious code in the cache. However, a more pressing concern is that because DMA approaches operate at a lower level than the kernel they do not have a clear view of dynamic kernel data structures, which requires that these structures need to be located in memory, a problem discussed in [81]. Even after locating the kernel data structures, there remains a synchronization issue between DMA operations and the host kernel: DMA cannot be used to acquire kernel locks on data structures. Consequently, race conditions result when the kernel is updating a data structure contemporaneous with a DMA read. False positives were observed by Baliga *et al*. [51] for precisely this reason. An inelegant solution [22] is to simply re-read memory locations containing suspicious values. Another consideration when implementing DMA approaches is the timing of DMA scans. Both [22] and [51] employed synchronous DMA scans, which are theoretically susceptible to timing attacks. Petroni *et al*. [22] suggested introducing randomness to the scan interval timings to overcome this susceptibility.

E. Virtualization Techniques

Summary: Use virtual environments to detect malware.

Advantages: Can be used to detect kernel-level rootkits and interpose state.

Disadvantages: Vulnerable to anti-emulation.

Virtualization, though technologically quite different from DMA, aims to satisfy the same goal of inspecting resources of the host machine without relying on the integrity of the operating system. Several techniques for rootkit detection, mitigation, and profiling that leverage virtualization have been developed, including [49], [72], [73], [82], [83]. The idea behind virtualization approaches is to involve a virtual machine monitor, a.k.a. the *hypervisor*, in the inspection of system resources. Since the hypervisor resides at a higher level of privilege than the guest OS, either on the hardware itself or simulated in software, and the hypervisor controls the access of the guest OS to hardware resources, the hypervisor can be used to inspect these resources even if the guest OS is entirely compromised. Unlike Copilot's approach, in which kernel writes and DMA reads are unsynchronized, the hypervisor and the guest OS kernel are synchronous since the guest OS relies on the hypervisor for resources. Moreover, the hypervisor has access to state information in the CPU, meaning that it can interpose state, a valuable ability not only for rootkit detection, prevention and mitigation, but also for computer forensics. Additionally, the hypervisor can be used to enforce site specific hardware policies, for example the hypervisor can prevent promiscuous mode network interface operation [72]. Hypervisors themselves may be vulnerable to attack, but the threat surface is much smaller than for an operating system: hypervisors have been written in as little as 30,000 lines of C code as opposed to the tens of millions of lines of code in modern Windows and Linux distributions. Significant security validations on hypervisors have also been conducted by academia, private security firms, the open source community, and intelligence organizations (e.g., CIA, NSA) [72].

Garfinkel and Rosenblum [72] created Livewire, a proof of concept intrusion detection system residing at the hypervisor layer. The authors refer to their approach as virtual machine introspection since the design utilizes an OS interface to translate raw hardware state into guest OS semantics and inspect guest OS objects via a policy engine, which interfaces with the view presented by the translation engine. The policy engine effectively is the intrusion detection system, which performs introspection on the virtual machine. The policy engine can monitor the machine and can also take mitigation steps such as pausing the state of the VM upon certain events or denying access to hardware resources.

A particular advantage of virtualization is that it can be leveraged to prevent rootkits from executing code in kernel memory – a task that all kernel rootkits must perform to load themselves into memory in the first place [82]. This includes DKOM rootkits: although the changes to kernel objects themselves cannot be detected as code changes to the kernel, code must be introduced at some point to make these changes. To this end, Seshadri *et al.* [82] formulated SecVisor. In contrast to the software-centric approach of Livewire, SecVisor leverages hardware support for virtualization of the x86 instruction set architecture as well as AMD's secure virtual machine technologies. SecVisor intercepts code via modifications to the CPU's memory management unit (MMU) and the I/O memory management unit (IOMMU), so that only code conforming to a user supplied policy will be executable. As such, kernel code violating the policy will not run on the hardware. In fact, SecVisor's modification to the IOMMU even protects the kernel from malicious writes via a DMA device. SecVisor works by allowing transfer of control to kernel mode only at entry points designated in kernel data structures, then performing comparisons to shadow copies of entry point pointers. This approach is analogous to that used in memory integrity checking modules of heavyweight dynamic binary instrumentation (DBI) frameworks like Valgrind [84].

Unfortunately, SecVisor has several drawbacks. First, modern Linux and Windows distributions mix code and data pages [83], while SecVisor's approach – enforcing *write XOR execute* ($W \oplus X$) permissions for kernel code pages through hardware virtualization – assumes that kernel code and data are not mixed within memory pages. The approach also fails for pages that contain self-modifying kernel code. Second, SecVisor requires modifications to the kernel itself – a difficult proposition for adoption on closed-source operating systems like Windows.

Riley *et al.* [83] formulated NICKLE (No Instruction Creeping into Kernel Level Executed), which, like SecVisor, leverages virtualization to prevent execution of malicious code in kernel memory. NICKLE approaches the problem via software virtualization and overcomes some of the limitations of SecVisor. NICKLE works by shadowing every byte of kernel code in a separate shadow memory writable only by the hypervisor. Because the hypervisor resides in a higher privilege domain than the kernel, even the kernel cannot modify the shadowed code. The shadowed code gets authenticated either during bootstrapping, when the kernel is loaded into memory, or when drivers are mounted or unmounted. Authentication consists of cryptographic hash comparisons of code segments with known good values taken by OS vendors or distribution maintainers. When the operating system requires access to kernel-level code, an indirection mechanism in the hypervisor reroutes this request to shadow values. To maintain transparency to the guest OS, this guest memory address indirection is implemented after the "virtual to physical" address translation in the hypervisors MMU. When the guest VM attempts to execute kernel code, a comparison is made to shadow memory. If the code is the same, then the shadow memory copy is executed. If the kernel memory and shadow memory code differ then one of several responses can be taken including logging and observing – an approach extended by Riley *et al.* [85] for rootkit profiling – rewriting the malicious kernel code with shadow values and continuing execution, or breaking execution. NICKLE's approach has two key advantages over SecVisor: first, it does not assume homogeneous code and data pages. Second, it does not require any modifications to kernel code. These benefits, however, incur hits in speed due to software virtualization and memory indirection costs and require a two-fold increase in memory for kernel code [83]. An additional complication arises from code relocation: when driver code is relocated in kernel memory, cryptographic hashes change. Riley *et al.* [83] handle this problem by tracking and ignoring relocated segments. Also, the NICKLE implementation in [83] does not support kernel page swapping, which would need to ensure that swapped in pages had the same cryptographic hash as when they were swapped out. Finally, NICKLE is ineffective in protecting self-modifying kernel code, a phenomenon present in both Linux and Windows kernels.

Srivastava *et al.* [49] leverage virtualization in their implementation of Sherlock – a defense system against the Illusion attack mentioned in Section II-B3. Sherlock uses the Xen hypervisor to monitor system call execution paths. Specifically, the guest OS is assumed to run on a virtual machine controlled by the Xen hypervisor. Monitoring of memory is conducted by the hypervisor itself with the aid of a separate security VM for system call reconstruction, analysis, and notification of other intrusion detections systems. Watchpoints are manually and strategically placed in kernel memory off-line, and a Büchi automaton [49] is constructed, which efficiently describes the expected and unexpected behavior of every system call in terms of watch points. Each watch point contains the `VMCALL` instruction, so that when it is hit, it notifies the hypervisor. Watch point identifiers are passed to the automaton as they are executed. During normal execution, the automaton remains in benign states and watch points are discarded. When a malicious state is reached, the hypervisor logs watch points and suspends the state of the guest VM. The function specific parameters at each watch point corresponding to a malicious state are then passed to the security VM for further analysis. An important consideration of this implementation is where to place watchpoints to balance effectiveness and efficiency. Srivastava *et al.* [49] manually chose watch point locations based on a reachability analysis of a kernel control flow graph, but suggest that an autonomous approach [86] could be implemented.

IV. Pattern-Based Stealth Malware Countermeasures

Pattern-based approaches aim to achieve more generic recognition of heterogeneous malwares. While these approaches offer potential for efficient updates and scalability beyond most component protection techniques, their increased generalization causes them to tend to exhibit higher recall rates but lower precision rates. Pattern-based approaches can be applied on static code fragments or on dynamic memory snapshots or behavioral data (e.g., system/API calls, network connections, CPU usage, etc.) and may be coupled with component protection approaches [26], [59], [87]. Static analysis has the advantage that it is fast [26], since the raw code is inspected but not executed; there is no need for an emulated environment. However, dynamic code mutation mechanisms outlined in Section II-E, are often able to hide functionalities from static code analyzers [87], and obfuscated code recognition techniques discussed in Section IV-C often rely on an emulated environment for decryption. Dynamic analysis tools that leverage emulated environments (see Section II-F) are not fooled so easily, since much of the underlying code, data, and behavior of the malware is revealed. However, dynamic analysis is potentially vulnerable to anti-emulation techniques (see Section II-F). While dynamic analysis techniques generally exhibit lower false-positive rates than static analysis techniques [87] dynamic techniques are far slower than static approaches [26] due to the need for an emulated environment, and can only be feasibly executed on a small number of code samples for short amounts of time. Consequently, hybrid approaches [59] are often employed, in which static methods are used to select suspicious samples, which are then passed to a dynamic analysis framework for further categorization.

A. Signature Analysis

Summary: Compare code signatures with database of malicious signatures via exact-matching or machine-learnt techniques.

Advantages: Detects known malwares reliably.

Disadvantages: Difficult to detect novel malware types.

Code-signature-based malware defenses are techniques that compare malware signatures – fragments of code or hashes of fragments of code – to databases of signatures associated with known attacks. Although signatures cannot be directly used to discover new exploits [76], they can do so indirectly due to component overlap between malwares [88], [89]. Ironically, shared stealth components have sometimes given away the presence of malwares that would have otherwise gone unnoticed [26]. Moreover, some byte sequences of length n (n-grams) specific to a common type of exploit are often present even under metamorphism of the code. Machine learning approaches to malware classification via n-gram and sequence analysis have been widely studied and deployed as integral components of anti-malware systems for more than ten years [26], [80].

While most in-memory rootkit signature recognition strategies behave much like on-disk signature strategies for detecting and classifying malicious code by matching raw bytes against samples from known malware, DKOM rootkit detection requires a different approach. Since DKOM involves changing existing data fields within OS data structures to hide them from view of certain parts of the OS, DKOM signature scanning techniques instead perform memory scans using signatures designed to pinpoint hidden data structures in kernel memory. Surprisingly, memory signature scans are useful both in live and forensics contexts. Chow *et al.* [90] demonstrated that structure data in kernel memory can survive up to 14 days after de-allocation, provided that the machine has not been rebooted. Schuster [91] formulated a series of signature rules for detecting processes and threads in memory, for the general purpose of computer forensics. Several spinoffs of this approach have been implemented. Unfortunately, many of these signature approaches can be subverted by rootkits that change structure header information. Dolan-Gavitt *et al.* [81] employed an approach to automatically obtain signatures for kernel data structures based on values in the structures that, if modified, cause the OS to crash. The approach includes data structure profiling and fuzzing stages. In the profiling stage, a clean version of the operating system is run, while a variety of tasks are performed. Kernel data structure fields commonly accessed by the OS are logged. The goal of the profiling stage is to determine fields that the OS often accesses and weed out fields that are not widely used for consideration as signatures. The fuzzing stage consists of running the OS on a virtual machine, pausing execution, and modifying the values in the candidate structure. After resuming, candidate structure values are added to the signature list if they cause the kernel to crash. The approach in [81] is in many ways the complement of the kernel invariant approach in [51]. Instead of traversing kernel data structures and examining which invariants are violated, Dolan-Gavitt *et al.* [81] scan all of kernel memory for plausible data structures. If certain byte offsets within the detected structures do not contain signatures consistent with certain values, then the detections cannot correspond to actual data structures used by the kernel because otherwise they would crash the operating system. A limitation of the approach in [81] is that it is susceptible to attack by scattering copies or near copies of data structures throughout kernel memory.

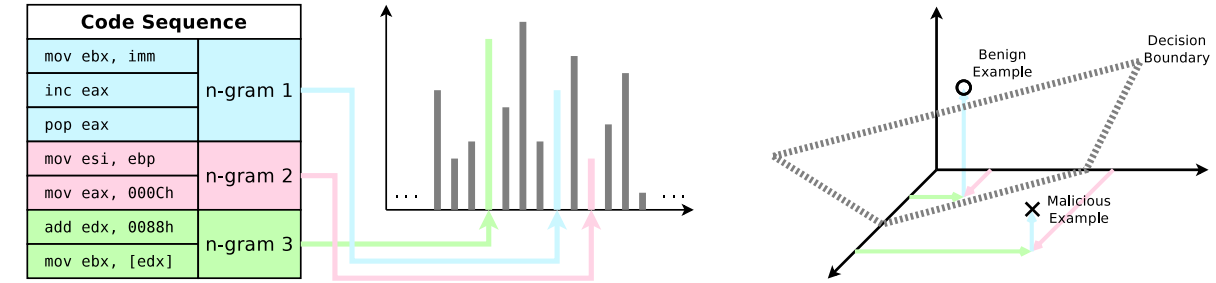
B. Behavioral/Heuristic Analysis

Summary: Derived from system behavior rather than code fragments.

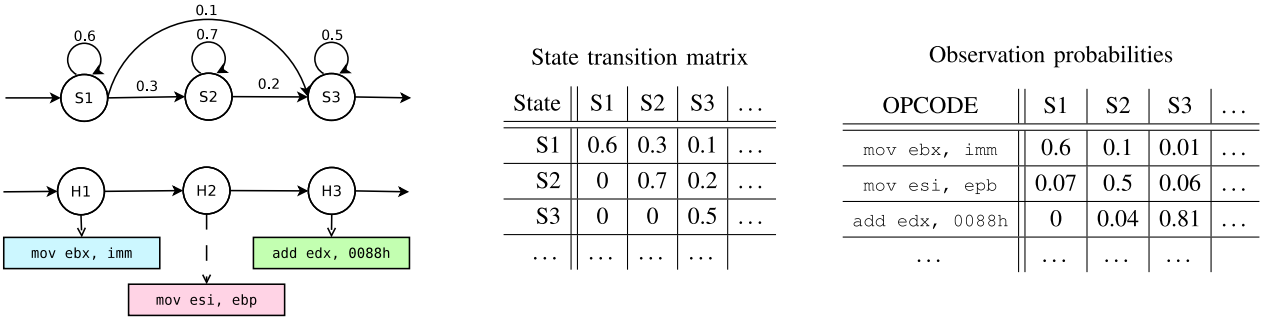
Advantages: Not affected by attempts to hide malicious code.

Disadvantages: Cannot detect malware prior to execution.

On the host level, signatures are not the only heuristic used for intrusion detection. System call sequences for intrusion and anomaly detection [92]–[98] are an especially popular alternative for rootkit analysis since hooked `IAT` or `SSDT` entries often make repetitive patterns of system calls. Interestingly, rootkits can also be detected by network intrusion detection systems (NIDSs) because rootkits in the wild are almost always small components of a larger malware. The larger malware often performs some sort of network activity such as C2 server communication and synchronization across infected



(a) Binary OPCODE n-gram histogram classification



State transition matrix

State	S1	S2	S3	...
S1	0.6	0.3	0.1	...
S2	0	0.7	0.2	...
S3	0	0	0.5	...
...	...	...	...	

Observation probabilities

OPCODE	S1	S2	S3	...
mov ebx, imm	0.6	0.1	0.01	...
mov esi, epb	0.07	0.5	0.06	...
add edx, 0088h	0	0.04	0.81	...
...	...	...	...	

(b) OPCODE sequence analysis

Fig. 6. Feature Space vs. State Space OPCODE classification. This diagram depicts (a) a schematic interpretation of a linear classifier that separates benign and malicious OPCODE n-grams in feature space and (b) a sequential OPCODE analysis using a hidden Markov model. The feature space model must explicitly treat histograms of n-grams as independent dimensions for varying values of n in order to capture sequential relationships. This approach is only scalable to a few sequence lengths. HMMs, on the other hand, impose a Markov assumption on a sequence of hidden variables which emit observations. State transition and observation probability matrices are inferred via expectation maximization on training sequences. An HMM factor graph is shown on the bottom left.

machines, or infection propagation. This is even true for some of the most sophisticated stealth malwares that leverage rootkit technologies to hide network connections from the host [60], while a rootkit cannot hide connections from a network. Therefore, signature scans at the network level as well as traffic flow analysis techniques can give away the presence of the larger malware as well as the underlying rootkit. Botnets with rootkits that effectively hide the behavior of an individual host may be easier to detect when analyzing macro, network-level traffic [99], [100]. NIDSs also have the advantage that they provide isolation between the malware and the intrusion detection system, reducing a malware's capacity to spoof or compromise the IDS. However, NIDSs have no way of inspecting the state of a host or interposing a host's execution at the network level. A hybrid approach, which extends the concept of cross-view detection is to compare network connections from a host query with those detected at the network level [101]. A discrepancy indicates the presence of a rootkit.

C. Feature Space Models vs. State Space Models

Summary: Classify code sequences.
Advantages: Can detect similar malicious code patterns.
Disadvantages: Cannot detect unknown or previously unseen malicious code.

As discussed above, code signatures and application behaviors/heuristics can be used in a variety of ways to detect and classify intrusions, and they operate across many levels of the intrusion detection hierarchy. For example, encrypted viruses are particularly robust against code signatures – until they are decrypted – but during emulation, once the virus is in memory, it might be particularly susceptible to signature analysis. This analysis may range from a simple frequency count of OPCODES to more sophisticated machine-learning techniques.

Machine learning models can be divided into feature space and state space models. Examples of both are shown in Fig. 6, in which code fragments are classified as malicious or benign based on their OPCODE n-grams. Feature space models aim to treat signature/behavioral features as a spatial dimension and parameterize a manifold within this high-dimensional feature space for each class. Feature space models can be further broken down into generative and discriminative models. Generative models aim to model the joint distribution $P(x, y)$ of target variable y and spatial dimension x, and perform classification via the product rule of probability: $P(y|x) = \frac{P(x,y)}{P(x)}$. Discriminative classifiers aim to model $P(y|x)$ directly [102]. By treating the frequencies of distinct n-gram hashes as elements of a high-dimensional feature vector, for example, the *input feature space* becomes the domain of these vectors. Support vector machines (SVMs), which are discriminative feature space classifiers, aim to separate classes by fracturing the input feature space (or some transformation thereof) by a hyperplane that maximizes soft class margins. An advantage of feature space models is that in high dimensions, even if only a few of the dimensions are relevant, different classes of data tend to separate [102]. However, feature space models do not explicitly account for probabilistic dependencies, and a good feature space from a classification accuracy perspective is not necessarily intuitive.

State space models are used to infer probabilities about *sequences*. They leverage the fact that certain sequences of instructions exist within malicious binary due to functional overlap as well as general lack of creativity and laziness of malware authors. State space models can also be applied to functional sequences (e.g., sequences of system calls or network communications). The intuition is that we can use certain types of functional behaviors to describe classes of malware in terms of what they do, for example, ransomwares like CryptoLocker typically generate a key that they use to encrypt files on disk and subsequently attempt to send that key to a C2 server. After a certain amount of time, they remove the local copy of the key and generate a ransom screen demanding money for the key [103]. State space models for intrusion recognition aim to recognize these sorts of malicious sequences.

The most common type of state space models are based in some form on the Markov assumption – that recent events will be independent of events that happened in the far past. While the Markov assumption is not always valid, it makes sequential inference tractable and is often reasonable. For example, if the last fifty assembly instructions were devoted to adding elements from two arrays together and incrementing respective pointers, with no other knowledge, it is a reasonable assumption that the next few instructions will add array elements. On the other hand, knowing that "hello world" was printed to the screen a million instructions ago provides little information about the probability of the next instruction. Hidden Markov models (HMMs) are perhaps the most widely used type of Markov models [102] and have been particularly useful in code analysis including recognition of metamorphic viruses [55], [104]–[111]. HMMs assume that latent variables, which take on *states*, are linked in a Markov chain with conditional dependencies on the previous states. The *order* of the HMM corresponds to the number of previous states on which the current state depends, for example, in an n-th order HMM the current state depends only on the previous n states.

In HMMs, previous states are fused with current states via a transition probability matrix A governing the Markov chain, and an observation probability matrix B – the probability of observing the data in a given state – as well as an initial state vector π. A, B, and π can be estimated via expectation maximization (EM) inference on observation sequences O, which aims to find the maximum likelihood estimate (MLE) of a sequence of observations, i.e., $\arg\max_{\lambda} P(O|\lambda)$, where $\lambda = (A, B, \pi)$. Although EM is guaranteed to converge to a local likelihood maximum, it is not guaranteed to converge to the global optimum. In the context of HMMs, this inference is usually carried out via the Baum-Welch algorithm [102] (aka. the Forward-Backward algorithm), which iterates between forward and backward passes and an update step until the likelihood of the observed sequence O is maximized with respect to the model.

The usage of HMMs for metamorphic virus detection has been documented in [55] and [104]–[111].[2] These works assume a predominantly decrypted virus body, i.e., little to no encryption within the body to begin with, or that a previously encrypted metamorphic has been decrypted inside an emulator. The number of hidden states and therewith the state transition matrix is generally chosen to be small (2-3), while observation matrix is larger, with rows consisting of conditional probabilities of OPCODES for given states. For metamorphic detection, the semantic *meaning* of the states themselves is unclear as is the optimal number of hidden states – they only reflect some latent structure within the code. This contrasts with other applications of HMMs, for example, in handwriting sequence recognition, the latent structure behind a noisy scrawl of an "X" is the letter "X" itself; thus with proper training data there should be 26 latent variables (for the English alphabet) with transition probabilities corresponding to what one might expect from an English dictionary, e.g., a "T" $\rightarrow$ "H" transition is much more likely than a "T" $\rightarrow$ "X" transition.

A common metamorphic virus recognition measure is the thresholded negative log-likelihood probability per OPCODE [55], [106], [109] obtained from a *forward* pass on an HMM, i.e.,:

$$-\frac{\log(p(O_1, .., O_N, z_1, .., z_N))}{N},$$

where $O_1, \ldots, O_N$ are OPCODEs in an N-length program and $z_1, \ldots, z_N$ are the hidden variables. The per-opcode normalization is required because different programs have different lengths. Most of the HMMs used in these works are first-order HMMs, in which the state space probability distribution of hidden variable z_n is conditioned only on the value of z_{n-1} and the current observation. For a $k - th$ order HMM, the probability of z_n is conditioned on $z_{n-1} \ldots z_{n-k}$. However, the time complexity of HMMs increases exponentially with their order. Although in their works [55], [104]–[111] the authors claim that the number of hidden variables did not seem to make a difference, they might if higher-order Markov chains were used. As Lin and Stamp [107] discuss, one problem with HMMs is that it ultimately measures similarity between code sequences; if the inter-class to intra-class sequential variation is large enough due to some exogenous factor such as very similar non-viral code in train/test, then HMM readout may be error-prone.

V. Toward Adaptive Models for Stealth Malware Recognition

A large portion of the malware detected by both component protection and generic recognition techniques is previously observed malware with known signatures, deployed by *script kiddies* – attackers with little technical expertise that predominantly use pre-written scripts to propagate existing attacks [112]. Systems with up-to-date security profiles are not vulnerable to such attacks.

Sophisticated stealth malwares, on the other hand, have propagated undetected for long periods of time because they do not match known signatures, do not attack protected system components with previously seen patterns, and mask harmful behaviors as benign. To reduce the amount of time that these previously unseen stealth malwares spend propagating in the

[2]HMMs are used for many sequential learning problems and have several different notations. Here, we borrow notation from [55].

wild, component protection and generic recognition techniques alike must be able to quickly recognize and adapt to new types of attacks. Typically, it is slower to adapt component techniques than it is to adapt generic recognition techniques because new hardware and software are required. However, even more generic algorithmic techniques may take time to update and this must be factored into the design of an intrusion recognition system.

The choice of the algorithm for efficient updates is only one of several considerations that must be addressed in an intrusion recognition system. More elementary is how to autonomously make a decision that additional training data is needed and that the classifier *needs to be updated* in the first place. In short, an intrusion recognition system must be *adaptive* in order to efficiently mitigate the threat of stealth malware. It must also be *interpretable* to yield actionable information to human operators and incident response modules. Unfortunately, many systems proposed in the literature are neither adaptive nor interpretable. We have isolated six flawed modeling assumptions, which we believe must be addressed at the algorithmic level. We discuss these flawed assumptions in Section V-A, and propose an algorithmic framework for attenuating them Section V-C.

A. Six Flawed Assumptions

1) Intrusions Are Closed Set: Real intrusion recognition tasks have unseen classes at classification time. Neither all variations of malicious code nor all variations of benign behaviors can be known apriori. However, the majority of the intrusion recognition techniques cited in this paper implicitly assume that all classes seen at classification time are also present in the training set, evaluating recognition accuracy only for a fixed *closed set* of classes.

Consequently, good performance on IDS benchmarks does not necessarily translate into an effective classifier in a real application. In real *open set* scenarios, a classifier that is trained on M classes of interest, at classification time is confronted with instances of classes that are sampled from a distribution of nearly infinitely-many categories. Conventional classifiers are designed to separate classes from one another by dividing a hypothesis space into regions and assigning labels respectively. Effective classifiers roughly seek to approximate the Bayesian optimal classifier on the posterior probability $P(y_i|x; C_1, C_2, \ldots, C_M), i \in \{1, \ldots, M\}$, where x is a feature vector, y_i is a class label, and C_i is a particular known class. However, in the presence of Ω unknown classes U_n the optimal posterior model would become $P(y_i|x; C_1, C_2, \ldots, C_M, U_1, \ldots, U_\Omega)$. Unfortunately, our ability to model this posterior distribution is limited because $U_1, \ldots, U_\Omega$ are unknown. Mining negatives during training may help to define known but uninteresting classes (e.g., C_{M+1}), but it is impossible to span all negative space, and the costs of negative training become infeasible with increasing numbers of feature dimensions. Consequently, a classifier may label space belonging to C_i far beyond the support of the training data for C_i. This fundamental machine learning problem has been termed *open space risk* [113]. Worse yet, if probability calibration is used, x may be ascribed to C_i with high confidence as distance from the positive side of the decision boundary increases. Therefore, the optimal closed set classifier operating in an open set intrusion recognition regime is not only wrong, it can be wrong while being very confident that it is correct. An open set intrusion recognition system needs to separate M known classes from one another, but must also manage open space risk by labeling a decision as "unknown" when far from known class data. Problems with the closed set assumption as well as desirable open set behavior are shown in Fig. 7.

The binary intrusion recognition task, i.e., *intrusion detection* appears to be a two-class closed set problem. However, each label – *intrusion* or *no intrusion* – is respectively a meta-label applied to a collection of many subclasses. While some of the subclasses will naturally be known, others will not, and the problem of open space risk still applies.

2) Anomalies Imply Class Labels: The incorrect assumption that anomalies imply class labels is largely related to the closed set assumption, and it is implicit to *all* binary malicious vs. benign classification systems. Anomalies constitute data points that deviate from statistical support of the model in question. In the classification regime, anomalies are data points that are far from the class to which they belong. In the open set scenario, anomalies should be resolved by an operator, protocol, or other recognition modalities. Effective anomaly detection is necessary for open set intrusion recognition. Without it, the implicit assumption of an overly closed set can lead to undesirable classifications because it forces a decision to be made without support of statistical evidence. The conflation between anomaly and intrusion assumes that anomalous behavior constitutes intrusive behavior and that intrusive behavior constitutes anomalous behavior. Often, neither of these assumptions hold. Especially in large networks, previously unseen benign behavior is common: new software installations are routine, new users with different usage patterns come and go, new servers and switches are added, thereby changing the network topology, etc. Stealth malwares, on the other hand, are specifically designed to exhibit normal behavior profiles and are less likely to be registered as anomalies than many previously unseen benign behaviors.

3) Static Models Are Sufficient: In the anti-malware domain, the assumption of a static model, which is implicit to the closed set modeling assumption, is particularly insufficient because of the need to update new nominal behavior profiles and malicious code signatures. The attacks that a system sees *will* change over time. This problem is often referred to as *concept drift* in the incremental learning literature [114]. Depending on the model, the time required for a full batch retrain may not be feasible. A kth-order HMM with $k >> 1$, for example, may perform quite well for some intrusion recognition tasks, but at the same time may be expensive to retrain in terms of both time and hardware and may require enormous amounts of training data in order to generalize well. There is a temporal risk associated with the amount of time that it takes to update a classifier to recognize new malicious samples. Therefore, even if that classifier exhibits high accuracy, it may

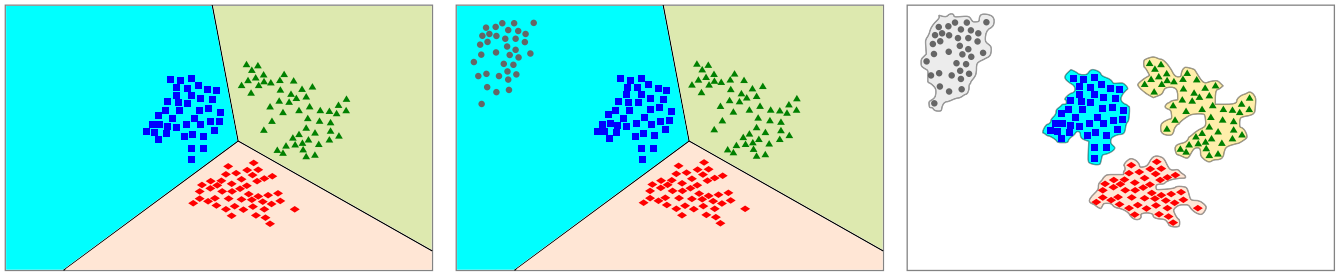


Fig. 7. Problems with the closed world assumption. (a) Red, green, and blue points correspond to a training set of different classes of malicious or benign samples in feature space. The intersecting lines depict a decision boundary learnt from training a linear classifier on this data. (b) The classifier categorizes points from a novel class (gray) as a training class (blue) with high confidence since the gray samples lie far on the blue side of the decision boundary and the classifier labels span infinitely in feature space. (c) An idealized open world classifier bounds the amount of space ascribed to each class' label by the support of the training data, labeling unlabeled (white) space as "unknown". With manually or automatically supplied labels, novel classes (gray) can be added to the classifier without retraining on the vast majority of data.

be vulnerable to temporal risk unless it possesses an efficient update mechanism.

4) No Feature Space Transformation Is Required: A key reason why machine learning algorithms are not overwhelmed by the curse of dimensionality is that, due to statistical correlations, classes of data tend to lie on manifolds that are highly non-linear, but effectively much smaller in dimension than the input space. Obtaining a good manifold representation via a feature transformation obtained from either hand-tuned or machine-learnt optimization is often critical to effective and discriminative classification. Many approaches in the intrusion detection literature simply pass raw log data or aggregated log data directly to a decision machine [115]–[121]. The inputs often possess heterogeneous scale and nominal and continuous-valued features with aggregations, which ignore temporal scale and varying spatial bandwidths. We contend that, like any other machine learning task, fine-grained discriminative intrusion recognition requires a meaningful feature space transformation, whether learnt explicitly by the classifier or carried out as a pre-processing task. Feature spaces for intrusion recognition have been explored [122]–[130]. While this research is a good start, we believe that much additional work is needed.

5) Model Interpretation Is Optional: Effective feature space transformations must be balanced with semantically meaningful interpretation. Unfortunately, these two objectives are sometimes conflicting. Neural networks, which have been successfully applied to intrusion recognition tasks [115], [131]–[136], are appealing because they provide the ability to adapt a fixed set of basis functions to input data, thus optimizing the feature space in which the readout layer operates. However, these basis functions correspond to a composition of aggregations of non-linear projections/liftings onto a locally optimal manifold prior to final readout, and neither the semantic meaning of the space, nor the semantic meaning of the final readout is well understood. Recent work has demonstrated that neural networks can be vulnerable to adversarial examples [137]–[139], which are misclassified with high confidence, yet appear very similar to known class data. The lack of interpretability of such models means that not only could intrusion recognition systems be vulnerable to such adversarial models, but more critically, machine learning techniques are not yet "smart" enough to resolve most potential intrusions. Instead, their role is to alert specialized anti-malware modules and human operators to take swift action. Fast response and resolution times are critical. Even if an intrusion detection system offers nearly perfect detection performance, if it cannot provide meaningful diagnostics to the operator, a temporal risk is induced, in which the operator or anti-malware wastes valuable time trying to diagnose the problem [1]. Also, as we have seen from previous sections, many potential malware signals (e.g., hooking) may stem from legitimate uses. It is important to know why an alarm was triggered and which features triggered it to determine and refine the system's response to both malicious and benign behaviors.

The interpretation and temporal risk problems are not unique to intrusion detection. They are a key reason why many diagnosis and troubleshooting systems rely on directed acyclic probabilistic graphical models such as Bayesian networks as well as rule mining instead of neural networks or support vector machines (SVMs) [140]. To better resolve the model interpretation problem, intrusion detection should move to a more generic recognition framework, ideally providing additional diagnostic information.

6) Class Distributions Are Gaussian: The majority of probabilistic models cited in this paper assume that class distributions are single or multi-modal Gaussian mixtures in feature space. Although Gaussian mixtures often appear to capture class distributions, barring special cases, they generally fail to capture distribution tails [141].

There are several different types of anomalies. *Empirical anomalies* are anomalous with respect to a probabilistic model of training data, whereas *idealized anomalies* are anomalous with respect to the joint distribution of training data. Provided good modeling, these two anomaly types are equivalent. However, from an anomaly detection perspective, naïve Gaussian assumptions do not provide a good match between empirical and idealized anomalies because an anomaly is defined with respect to the tail of a joint distribution and tails tend to deviate from Gaussian [141]. Theorems from statistical extreme value theory (EVT) provide theoretically grounded functional forms for the classes of distributions that these class-tails can assume, provided that positive class outliers

are *process anomalies* – rare occurrences from an underlying generating stochastic process – and not noise exogenous to the process, e.g., previously unseen classes.

B. An Open Set Recognition Framework

Accommodating new attack profiles and normative behavior models requires a method for diagnosing when query data are unsupported by previously seen training samples. This diagnosis is commonly referred to as *novelty detection* in [142]. Specifically in IDS literature, novelty detection is often hailed as a means of detecting malicious samples with no prior knowledge. The intuition is that by spanning the space of normal behavior during training, any novel behavior will be either an attack or a serious system error. In practice however, it is infeasible to span the space of benign behavior. Even on an individual host, "normal" benign behavior can change dramatically depending on configurations, software installations, and user turnover. The network situation is even more complicated. Even for a medium size network, services, protocols, switches, routers, and topologies vary routinely.

We contend that novelty detection has a particularly useful role in the recognition of stealth malware, but the premise that we can span the entire benign input space apriori is as unrealistic as the premise that signatures of all known attacks solve the intrusion detection problem. Instead, novelty detection should be treated in terms of what it does mathematically – as a tool to recognize samples that are unsupported by the training data and to quantify the degree of confidence to ascribe a model's decision. Specifically, we propose treating the intrusion recognition task as an *open set recognition* problem, performing discriminative multi-class recognition under the assumption of unknown classes at classification time. Scheirer *et al.* [113], [143] formalized the open set recognition problem as tradeoff between minimizing *empirical risk* and *open space risk* – the risk of labeling unknown space – or mathematically, the ratio of positively labeled space that should have been labeled "unknown" to the total extent of positively labeled space. A classifier that can arbitrarily control this ratio via an adjustable threshold is said to *manage open space risk*.

Scheirer *et al.* [113] extended the linear SVM objective to bound data points belonging to each known class by two parallel hyperplanes; one corresponding to a discriminative decision boundary, managing empirical risk, and the other limiting the extent of the classification, managing open space risk. Unfortunately, this "slab" model is not easily extensible to a non-linear classifier. In later work [143], [144], they extended their solution to multi-class open set recognition problems using non-linear kernels, via posterior EVT calibration and thresholding of nonlinear SVM decision scores. EVT-calibrated one-class SVMs are used in conjunction with multi-class SVMs to simultaneously bound open-space risk and provide strong discriminative capability [143]. The authors refer to this combination as the *W-SVM*. For our discussion, however, the theorems of Scheirer *et al.* [143] are more interesting than the W-SVM itself. They prove that sum, product, min, and max fusions of compact abating probability (CAP) models again generate CAP models. Bendale and Boult [145] extended this work to show that CAP models in linearly transformed spaces manage open space risk in the original input space. While these works are interesting, the formulations limit their application to probability distributions. Due to the need for efficient model updates in an intrusion recognition setting, enforcing probabilistic constraints on the recognition problem might be non-trivial, due to the need to re-normalize at each increment. We therefore generalize the theorems of Scheirer *et al.* [143] as follows.

Theorem 1 (Abating Bounds for Open Space Risk): Assume a set of non-negative continuous bounded functions $\{g_1, \ldots, g_n\}$ where $g_k(x, x')$ decreases monotonically with $||x - x'||$. Then thresholding any positively weighted sum, product, min, or max fusion of a finite set of non-negative discrete or continuous functions $\{f_1, \ldots, f_n\}$ that satisfy $f_k(x, x') \leq g_k(x, x')\ \forall k$ manages open space risk, i.e., it allows us to constrain open space risk below any given ϵ.

Proof: Given $\tau > 0$, define

$$g'_k(x, x', \tau) \coloneqq \begin{cases} g_k(x, x') & \text{if } g_k(x, x') > \tau \\ 0 & \text{otherwise.} \end{cases}$$

$$f'_k(x, x', \tau) \coloneqq \begin{cases} f_k(x, x') & \text{if } f_k(x, x') > \tau \\ 0 & \text{otherwise.} \end{cases}$$

This yields $f'_k(x, x') \leq g'_k(x, x')\ \forall k$. Because of the monotonicity of g_k, for any fixed constant δ, $\exists \tau_\delta : \int g'_k(x, x', \tau)\ dx \leq \delta$. Combining that with $f'_k(x, x') \leq g'_k(x, x')$, yields $\int f'_k(x, x', \tau_\delta)\ dx \leq \delta$, thus limiting positively labeled area to $f_k(x, x') > \tau$, which manages open space risk. Without loss of generality on k, it is easy to see that max and min fusion also manage open space risk. Because summation is a linear operator:

$$\int \sum_k f'_k(x, x', \tau)\ dx = \sum_k \int f'_k(x, x', \tau)\ dx.$$

Since a finite sum of finite values is finite, and $\sum_k \int f'_k(x, x', \tau)\ dx \leq k\delta$ it follows that thresholded positively weighted sums of f'_k manage open space risk. In addition, $\prod_k f'_k$ is bounded because:

$$g_k \Rightarrow \exists \eta\ :\ g'_k(x, x', \tau) < \eta \Rightarrow \int \prod_k g'_k(x, x', \tau) dx \leq \eta^k \delta.$$

This latter bound may not be tight, but is sufficient to show that $\prod_k g'_k(x, x', \tau)$ manages open space risk. We have proven Theorem 1 without weights in the sums and products, but without loss of generality, non-negative weights can be incorporated directly into g_k. ■

From Theorem 1, it directly follows that many *novelty detection* algorithms already in use by the IDS community provably manage open space risk and fit nicely into the open set recognition framework. For example, Scheirer *et al.* [143] prove that thresholding neighbor methods by distance manages open space risk. Via such thresholding, clustering methods can be extended to an online regime, in which unknown classes $U_1, \ldots, U_\Omega$ are isolated [142]. Similarly, thresholded kernel density estimation (KDE) of "normal" data distributions has been successfully applied to the IDS domain.

Yeung and Chow [146] used kernel density estimates, in which they centered an isotropic Gaussian kernel on every data point x_k. It is easy to prove that such estimates also manage open space risk.

Corollary 1 (Gaussian KDE Bounds for Open Space Risk): Assume a Gaussian kernel density estimator where:

$$p(x) = \frac{1}{N}\sum_{k=1}^{N}\frac{1}{\left(2\pi\sigma^2\right)^{D/2}}exp\left(\frac{\|x-x_k\|^2}{2\sigma^2}\right).$$

Thresholding $p(x)$ by $0 < \tau \leq 1$ manages open space risk.

Proof: When N is the total number of points, each kernel is given by:

$$f_k(x, x_k) = \frac{1}{N}\frac{1}{\left(2\pi\sigma^2\right)^{D/2}}exp\left(\frac{\|x-x_k\|^2}{2\sigma^2}\right).$$

By Theorem 1, we can treat $f_k(x, x_k)$ as its own bound. When thresholded, $f_k(x, x_k)$ will finitely bound open space risk. The kernel density estimate:

$$p(x) = \sum_{k=1}^{N} f_k(x, x_k) = \frac{1}{N}\sum_{k=1}^{N}\frac{1}{\left(2\pi\sigma^2\right)^{D/2}}exp\left(\frac{\|x-x_k\|^2}{2\sigma^2}\right)$$

also bounds open space risk because it is a positively weighted sum of functions that satisfy the bounding criteria in Theorem 1. ■

Thresholded nearest neighbor approaches and KDE require selection of a meaningful σ, and distance/probability threshold. They also implicitly assume local isotropy in the feature space, which highlights the need for a meaningful feature space representation.

Neighbor and kernel density estimators are nonparametric models, but several parametric novelty detectors in use by the IDS community also provably manage open space risk. Thresholding density estimates from Gaussian mixture models (GMMs) is a popular parametric approach to novelty detection with a similar functional form to KDE [147]–[151]. For GMMs, however, the input data x is assumed distributed as a superposition of a *fixed* number of Gaussians:

$$p(x) = \sum_{k} c_k \mathcal{N}(x|\mu_k, \Sigma_k)$$

such that $\sum_k c_k = 1$. Unlike nonparametric Gaussian KDE, in which μ and σ are selected apriori, the Gaussians in a GMM are fit via an expectation maximization technique similar to that used by HMMs. By generalizing Corollary 1, we can prove that thresholding GMMs probabilities manages open space risk. When GMMs integrate to one, they are also CAP models. Although this constraint often holds, it is not required.

Corollary 2 (GMM Bounds for Open Space Risk): Assume a Gaussian mixture model. The thresholded density estimate from this model bounds open space risk.

Proof: By Theorem 1, the kth mode of a GMM:

$$f_k(x, \mu) = c_k e^{-\frac{1}{2}(x-\mu)^T(x-\mu)}$$

is its own abating bound. Because the superposition of all modes is a sum of non-negatively weighted functions, each with an abating bound, GMMs have an abating bound. Thresholding GMM density estimates therefore manages open space risk. ■

Note that Corollary 2 only holds for the density estimate from an individual GMM, and not necessarily for all recognition functions that leverage multiple GMMs. For example, the log ratio of probabilities of two GMM estimates, $\log\frac{p_1(x)}{p_2(x)}$ does not bound open space risk when $\frac{p_1(x)}{p_2(x)}$ diverges as either $p_1(x)$ or $p_2(x) \to 0$. Similarly a recognition function $p_1(x) > p_2(x)$ does not provably manage open space risk because $p_1(x) > p_2(x)$ can hold over unbounded x.

There is a strong connection between GMMs and the aforementioned HMMs. Similarly to HMMs, GMMs can also be viewed as discrete latent variable models. Given input data x and multinomial random variable z, whose value corresponds to the generating Gaussian, the joint distribution factors according to the product rule: $p(x, z) = p(z)p(x|z)$. $p(z)$ is determined by the Gaussian mixture coefficients c_k. Therefore, the factorization of GMMs can be viewed as a simplification of HMMs, with a factor graph, in which latent variables are not connected and, therefore, treated independent of sequence.

This raises two questions: first, can HMMs can be used for novelty detection? And second, do HMMs manage open space risk? Indeed, HMMs *can* be used for novelty detection on sequential data by running inference on sequences in the training set and thresholding the estimated joint probability (or log of the estimated joint probability) outputs. This approach was taken by Yeung and Ding [152] for host-based intrusion detection using system call sequences. To assess, whether HMMs manage open space risk, we need to consider the form of an HMM's estimated joint distribution. For an N-length sequence, an HMM factors as:

$$p(x_1, .., x_N, z_1, .., z_N) = p(z_1)\prod_{n=2}^{N} p(z_n|z_{n-1})\prod_{n=1}^{N} p(x_n|z_n)$$

where $x_1, \ldots, x_N$ are observations and $z_1, \ldots, z_N$ are latent variables. This leads to Corollary 3.

Corollary 3 (HMM Bounds for Open Space Risk): Assume HMM factors $p(z_1)$, $p(z_n|z_{n-1})$, and $p(x_n|z_n)$ satisfy the bounding constraints in Theorem 1. Then thresholding the output of a forward pass of an HMM bounds open space risk.

Proof: Under the assumption that $p(z_1)$, $p(z_n|z_{n-1})$, and $p(x_n|z_n)$ satisfy the bounding constraints in Theorem 1, then the HMM factorization above is a product of these functions, which by Theorem 1 manages open space risk. ■

Corollary 3 states that under certain assumptions on the form of the factors in HMMs, an HMM will provably manage open space risk. Unfortunately, it is not immediately clear how to enforce such a form, so many HMMs, including those in [152] are not proven to manage open space risk and may ascribe known labels to infinite open space. Formulating HMMs that manage open space risk and provide adequate

modeling of data is an important topic, which we leave for future research.

GMMs and HMMs are linear models. One-class SVMs are popular nonlinear models, which have been successfully applied to detecting novel intrusions [153]–[158]. In their Theorem 2, Scheirer *et al.* [143] prove that one-class SVM density estimators [159] with a Gaussian radial-basis function (RBF) kernel manage open space risk. The decision functions for these machines are given by $\sum_k \alpha_k K(x, x_k)$, where $K(x, x_k)$ is the kernel function and α_i are the Lagrange multipliers. It is important to note that non-negative α_k are required to satisfy Theorem 1 in [143], and that multi-class RBF SVMs and one-class SVMs under different objective functions are not proven to manage open space risk.

C. Open World Archetypes for Stealth Malware Intrusion Recognition

The open set recognition framework introduced in Section V-B can be *incorporated* into existing intrusion recognition algorithms. This means that there is no need to abandon closed set algorithms in order to manage open space risk, provided that they are fused with open set recognition algorithms. Closed set techniques may be excellent solutions when they are well supported by training data, but open set algorithms are required in order to ascertain whether the closed set decisions are meaningful. Therefore, the open set problem can be addressed by using an algorithm that is inherently open set for novelty detection and rejecting any closed set decision as unknown if its support is below the open set threshold. A model with easily interpreted diagnostic information, e.g., a decision tree or Bayesian network, can be fused with the open set algorithm as well, in order to decrease response/mitigation times and to compensate for other discriminative algorithms that are not so readily interpretable. Note that many of the algorithms proposed by Scheirer *et al.* are discriminative classifiers themselves, but underperform the state of the art in a purely closed setting.

The interpretation of a thresholded open set decision is trivial, assuming that the recognition function represents some sort of density estimation. For a query point, if the maximum density with respect to each class is below the open set threshold, τ, then the class is labeled as "unknown". Otherwise, the query sample is ascribed the label corresponding to the class of maximum density. Under the open set formulation, the degree of *openness* can be controlled by the value of τ. The desired amount of openness will vary depending on the algorithm and the application's optimal precision-recall requirements. For example, a high security non-latency sensitive virtualized environment that is administered by many security experts can label many examples as unknown and interpose state frequently for an expert opinion. Systems that are latency sensitive, but for which potential harm of intrusion is relatively low, might have much looser open space bounds.

Note that an open set density estimator can be applied with or without normalization to a probability distribution. However, we can only prove that it manages open space risk if the estimator's decision function satisfies Theorem 1.

Open set algorithms can also be applied under many different feature space transformations. When open set algorithms are fused with closed set algorithms, the two need not necessarily operate in the same feature space. Research has demonstrated [145], [160] the effectiveness of the open set classification framework in machine-learnt feature spaces. Bendale and Boult [145] bounded a nearest class mean (NCM) classifier in a metric-learnt transformed space, an algorithm they dubbed nearest non-outlier (NNO). They also proved that under a linear transformation, open space risk management in the transformed feature space will manage open space risk in the original input space. Rudd *et al.* [160] formulated extreme value machine (EVM) classifiers to perform open set classification in a feature space learnt from a convolutional neural network. The EVM formulation performs a kernel-like transformation, which supports variable data bandwidths, that implicitly transforms *any* feature space to a probabilistically meaningful representation. This research indicates that open set algorithms can support meaningful feature space transformations, although what constitutes a "good" feature space depends on the problem and classifier in question.

Bendale and Boult [145] and Rudd *et al.* [160] also extended open set recognition algorithms to an online regime, which supports incremental model updates. They dubbed this recognition scenario *open world recognition*, referring to online open set recognition. The incremental aspects of this work are in a similar vein to other online intrusion recognition techniques [161]–[167], which, given a batch of training points X_t at time t, aim to update the prior for time $t+1$ in terms of the posterior for time t, so that $P_{t+1}(\theta_{t+1}) \leftarrow P_t(\theta_t|X_t, T_t)$, where T is the target variable, P is a recognition function, and θ is a parameter vector. If P is a probability, a Bayesian treatment can be adopted, where:

$$P_{t+1}(\theta_{t+1}|X_{t+1}, T_{t+1}) = \frac{P_{t+1}(T_{t+1}|\theta_{t+1}, X_{t+1})P_t(\theta_t|X_t, T_t)}{P_{t+1}(T_{t+1})}$$

With a few exceptions, however, recognition functions in the incremental learning intrusion recognition literature generally do not satisfy Theorem 1, and are not proven to manage open space risk. This means that they are not necessarily true *open world* classifiers.

Moreover, none of the work in [161]–[167] addresses the pressing need to *prioritize labeling of detected novel* data for incremental training. This is problematic because the objective of online learning is to adapt a model to recognize new attack variations and benign patterns – insights that would otherwise be perishable within a useful time horizon. When intrusion recognition subsystems exhibit high recall rates, however, updating the model with new attack signatures is *much more vital* than updating the model with novel benign profiles. Since labeling capacity is often limited by the number of knowledgeable security practitioners, we contend that the "optimal" labeling approach is to greedily rank the unknown samples in terms of their likelihood of being associated with known malicious classes. Given bounded radially abating functions from Theorem 1, i.e., open set decision functions, we can do

just that, prioritizing labeling by some malicious likelihood criterion (MLC).

The intuition behind the MLC ranking is as follows: from the discussion in Section IV, malwares often share components: even for vastly different malwares, similar components yield similar patterns in feature space. Although minor code overlap will not necessarily cause (mis)categorizations of malware classes, it may cause novel malware classes to be close enough to known ones in feature space that they are ranked higher by MLC criterion than most novel benign samples. Label prioritization by MLC ranking could, therefore, improve resource allocation of security professionals and dramatically reduce the amount of time that stealth malwares are able to propagate unnoticed. Of course, other considerations besides MLC are relevant to a truly "optimal" ranking, including difficulty of diagnosis and likely degree of harm, but these properties are difficult to ascertain autonomously.

A final useful aspect of the open world intrusion recognition framework is that it is not confined to naive Gaussian assumptions. Mixtures of Gaussians can work well for modeling densities, but tend to deteriorate at the distribution tails, because the tails of the models tend toward tails from unimodal Gaussians, whereas the tails of the data distributions generally do not. For recognition problems, however, accurate modeling of tail behavior is important, in fact, more important than accurate modeling of class centers [168], [169]. To this end, researchers have turned to statistical extreme value theory techniques for density estimation, and open world recognition readily accommodates them. Both [144] and [143] apply EVT modeling to open set recognition scenarios based posterior fitting of point distances to classifier decision boundaries, while [160] incorporated EVT calibration into a generative model, which performs loose density estimation as a mixture of EVT distributions. Importantly, the EVT distributions employed by Rudd *et al.* [160], unlike Gaussian kernels in an SVM or KDE application, are variable bandwidth functions of the data. They are also are directly derived from EVT and incorporate higher-order statistics which Gaussian distributions cannot (e.g., skew, curtosis). Finally, they provably manage open space risk.

VI. Conclusions and Open Issues

Stealth malwares are a growing threat because they exploit many system features that have legitimate uses and they can propagate undetected for long periods of time. We, therefore, felt the need to provide the first academic survey specifically focused on malicious stealth technologies and mitigation measures. We hope that security professionals in both academic and industrial environments can draw on this work in their research and development efforts. Our work also highlights the need to combine countermeasures that aim to protect the integrity of system components with more generic machine learning solutions. We have identified flawed assumptions behind many machine learning algorithms and proposed steps to improve them based on research from other recognition domains. We encourage the security community to consider these suggestions in future development of intrusion recognition algorithms.

While we are the first to propose a mathematically formalized *open world* approach to intrusion recognition, there are open issues that must be addressed through experimentation and implementation, including how tightly to bound open space risk, and more generally how to determine the openness of the problem in operational scenarios. An overly aggressive bound may actually degrade performance for problems that are predominantly closed-set, prioritizing the minimization of open space risk over the minimization of empirical risk. Another important consideration is the cost of misclassifying an unknown sample as belonging to a known class, which depends in part on the operational resources available to label novel classes, and in part on the degree of threat expected of novel classes. These tradeoffs are important subjects for experimental analysis and operational deployment of open world anti-malware systems.

For benchmarking and experimentation, good datasets that support open world protocols are vital for future research. While some effort has been made, e.g., [170]–[172], there are few modern publicly available datasets for intrusion detection, specifically of stealth malwares. We believe that the collection and distribution of modernized and realistic publicly available datasets containing stealth malware samples are vital to the furtherance of academic research in the field. While many corporate security companies have good reasons for keeping their datasets private, a guarded increase in collusion with academia to allow extended – yet still restricted – sharing of data is in the best interest of all parties in developing better stealth malware countermeasures.

We have proven that a number of existing algorithms currently used in the intrusion recognition domain already satisfy the requirements of an open set framework, and we believe that they should be leveraged and extended both in theory and in practice to address the flawed assumptions behind many existing algorithms that we detailed in Section V-A. Adopting an open world mathematical framework obviates the assumptions that *intrusions are closed set*, *anomalies imply class labels*, and that *static models are sufficient*. How to appropriately address the other assumptions requires further research. Although some progress has been made in open world algorithms, the question, how to obtain a nicely discriminable feature space while accommodating a readily interpretable model, merits future research. Finally, how to model class distributions without Gaussian assumptions demands further mathematical treatment – statistical extreme value theory is a good start, but it has yet to be gracefully defined how select distributional tail boundaries. Also, with the exception of special cases it is still not well formalized, how to model the remainder of the distribution (the non-extreme values) of non-Gaussian data.

Appendix
API Calls, Data Structures, and Registry Keys

See Table I.

TABLE I
SYSTEM CALLS. THIS TABLE EXPLAINS THE WINDOWS SYSTEM CALLS, DATA STRUCTURES, REGISTRY KEYS AND SYSTEM FILES (IN THIS ORDER) THAT ARE USED IN THE MALWARE DESCRIBED IN THIS SECTION, IN ALPHABETICAL ORDER. MANY OF THE ENTRIES ARE COPIED DIRECTLY FROM THE MICROSOFT DEVELOPER NETWORK (MSDN) DOCUMENTATION [173] OR WIKIPEDIA FOR FILE DESCRIPTIONS [174]. OTHERS ARE SUMMARIES OF DESCRIPTIONS FROM LATER IN THE TEXT WITH THEIR OWN RESPECTIVE CITATIONS

Windows API entry	**Documentation**
`CallNextHookEx`	Passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.
`CreateRemoteThread`	Creates a thread that runs in the virtual address space of another process.
`DeviceIoControl`	Sends a control code directly to a specified device driver, causing the corresponding device to perform the corresponding operation.
`DllMain`	An optional entry point into a dynamic-link library (DLL). When the system starts or terminates a process or thread, it calls the entry-point function for each loaded DLL using the first thread of the process. The system also calls the entry-point function for a DLL when it is loaded or unloaded using the `LoadLibrary` and `FreeLibrary` functions.
`FindFirstFile`	Searches a directory for a file or subdirectory with a name that matches a specific name (or partial name if wildcards are used).
`FindNextFile`	Continues a file search from a previous call to the `FindFirstFile`, `FindFirstFileEx`, or `FindFirstFileTransacted` functions.
`GetProcAddress`	Retrieves the address of an exported function or variable from the specified dynamic-link library (DLL).
`LoadLibrary`	Loads the specified module into the address space of the calling process. The specified module may cause other modules to be loaded.
`NtDelayExecution`	*Undocumented export of `ntdll.dll`.*
`NTQuerySystemInformation`	Retrieves the specified system information.
`OpenProcess`	Opens an existing local process object.
`PsSetLoadImageNotifyRoutine`	The `PsSetLoadImageNotifyRoutine` routine registers a driver-supplied callback that is subsequently notified whenever an image is loaded (or mapped into memory).
`PspCreateProcessNotify`	*Completely undocumented function.*
`SetWindowsHookEx`	Installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated either with a specific thread or with all threads in the same desktop as the calling thread.
`SleepEx`	Suspends the current thread until the specified condition is met.
`WriteProcessMemory`	Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.
`ZwQuerySystemInformation`	Retrieves the specified system information.
`EAT`	The `Export Address Table` is a table where functions exported by a module are placed so that they can be used by other modules.
`EPROCESS`	The `EPROCESS` structure is an opaque executive-layer structure that serves as the process object for a process.
`IAT`	The `Import Address Table` is where the dynamic linker writes addresses of loaded modules such that each entry points to the memory locations of library functions.
`IDT`	The `Interrupt Descriptor Table` is a kernel-level table of function pointers to callbacks that are called upon interrupts or exceptions.
`IRP`	`I/O Request Packets` are used to communicate between device drivers and other areas of the kernel.
`KTHREAD`	The `KTHREAD` structure is an opaque kernel-layer structure that serves as the thread object for a thread.
`KeServiceDescriptorTable`	Contains pointers to the `SSDT` and `SSPT`. It is an undocumented export of `ntoskrnl.exe`.
`SSDT`	The `System Service Descriptor Table` is a kernel-level dispatch table of callbacks for system calls.
`SSPT`	The `System Service Parameter Table` is a kernel-level table containing sizes (in bytes) of arguments for `SSDT` callbacks.
`AppInit_DLLs`	Space or comma delimited list of DLLs to load.
`LoadAppInit_DLLs`	Globally enables or disables `AppInit_DLLs`.
`kernel32.dll`	Exposes to applications most of the Win32 base APIs, such as memory management, input/output (I/O) operations, process and thread creation, and synchronization functions.
`ntdll.dll`	Exports the Windows Native API. The Native API is the interface used by user-mode components of the operating system that must run without support from Win32 or other API subsystems.
`ntoskrnl.exe`	Provides the kernel and executive layers of the Windows NT kernel space, and is responsible for various system services such as hardware virtualization, process and memory management, thus making it a fundamental part of the system.
`user32.dll`	Implements the Windows user component that creates and manipulates the standard elements of the Windows user interface, such as the desktop, windows, and menus.

REFERENCES

[1] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, USA, 2010, pp. 305–316.

[2] W.-X. Li, J.-B. Wang, D.-J. Mu, and Y. Yuan, "Survey on android rootkit," *CNKI—Microprocess.*, vol. 32, no. 2, pp. 68–72, 2011.

[3] S. Kim, J. Park, K. Lee, I. You, and K. Yim, "A brief survey on rootkit techniques in malicious codes," *J. Internet Services Inf. Security*, vol. 3, no. 4, pp. 134–147, 2012.

[4] T. Shields. (2008). *Survey of Rootkit Technologies and Their Impact on Digital Forensics*. [Online]. Available: http://www.donkeyonawaffle.org/misc/txs-rootkits_and_digital_forensics.pdf

[5] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Dept. Comput. Eng., Chalmers Univ. Technol., Gothenburg, Sweden, Tech. Rep., 2000.

[6] E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer, "Taxonomy and survey of collaborative intrusion detection," *ACM Comput. Surveys*, vol. 47, no. 4, pp. 1–33, 2015.

[7] R. Zuech, T. M. Khoshgoftaar, and R. Wald, "Intrusion detection and big heterogeneous data: A survey," *J. Big Data*, vol. 2, no. 1, pp. 1–41, 2015.

[8] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, "Intrusion detection by machine learning: A review," *Expert Syst. Appl.*, vol. 36, no. 10, pp. 11994–12000, 2009.

[9] P. Garcia-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *Comput. Security*, vol. 28, nos. 1–2, pp. 18–28, 2009.

[10] W. Lee, S. J. Stolfo, and K. W. Mok, "A data mining framework for building intrusion detection models," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, USA, 1999, pp. 120–132.

[11] "Kaspersky security bulletin," Kaspersky Lab, Moscow, Russia, Tech. Rep., 2015.

[12] "HPE security research cyber risk report," Hewlett Packard Enterprise, Palo Alto, CA, USA, Tech. Rep., 2015.

[13] "HPE security research cyber risk report," Hewlett Packard Enterprise, Palo Alto, CA, USA, Tech. Rep., 2016.

[14] IBM X-Force, "IBM X-force threat intelligence report 2016," IBM, Armonk, NY, USA, Tech. Rep., 2016.

[15] "Internet security threat report," Symantec, Mountain View, CA, USA, Tech. Rep., Apr. 2015.

[16] "Internet security threat report," Symantec, Mountain View, CA, USA, Tech. Rep., Apr. 2016.

[17] "McAfee labs threats report," Intel Security, Santa Clara, CA, USA, Tech. Rep., Mar. 2016.

[18] "Microsoft security intelligence report," Microsoft Corporat., Redmond, WA, USA, Tech. Rep., Dec. 2015.

[19] "M-Trends," Mandiant Consult., Alexandria, VA, USA, Tech. Rep., Feb. 2016.

[20] P. Faruki *et al.*, "Android security: A survey of issues, malware penetration, and defenses," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 2, pp. 998–1022, 2nd Quart., 2015.

[21] A. Chuvakin, "Ups and downs of UNIX/Linux host-based security solutions," *Login: Mag. USENIX SAGE*, vol. 28, no. 2, pp. 57–62, Apr. 2003. [Online]. Available: https://www.usenix.org/publications/login/april-2003-volume-28-number-2/ups-and-downs-unixlinux-host-based-security

[22] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot—A coprocessor-based kernel runtime integrity monitor," in *Proc. 13th Conf. USENIX Security Symp. (SSYM)*, Berkeley, CA, USA, 2004, p. 13. [Online]. Available: http://www.jesusmolina.com/publications/2004NPTF.pdf

[23] J. Butler and S. Sparks, "Windows rootkits of 2005, part one," Symantec Connect Community, Tech. Rep., 2005. [Online]. Available: http://www.symantec.com/connect/articles/windows-rootkits-2005-part-one

[24] G. H. Kim and E. H. Spafford, "The design and implementation of tripwire: A file system integrity checker," in *Proc. 2nd ACM Conf. Comput. Commun. Security*, Fairfax, VA, USA, 1994, pp. 18–29.

[25] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Upper Saddle River, NJ, USA: Addison-Wesley, 2005.

[26] P. Szor, *The Art of Computer Virus Research and Defense*. Upper Saddle River, NJ, USA: Addison-Wesley, 2005.

[27] E. Rodionov and A. Matrosov. *Mind the Gapz: The Most Complex Rootkit Ever Analyzed?* [Online]. Available: http://www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf

[28] A. Matrosov. (Oct. 2012). *Olmasco Bootkit: Next Circle of TDL4 Evolution (or Not?)* [Online]. Available: http://www.welivesecurity.com/2012/10/18/olmasco-bootkit-next-circle-of-tdl4-evolution-or-not-2/

[29] J. Butler and G. Hoglund, "VICE—Catch the hookers!" presented at the Black Hat USA, 2004. [Online]. Available: http://www.infosecinstitute.com/blog/butler.pdf

[30] Microsoft Support, "What is a DLL?" 2007, Art. no. 815065. [Online]. Available: http://support.microsoft.com/kb/815065

[31] J. Leitch, "IAT hooking revisited," 2011.

[32] *Introduction to x64 Assembly Intel Software*. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-x64-assembly

[33] Microsoft Research. (1999). *Detours*. [Online]. Available: http://research.microsoft.com/en-us/projects/detours

[34] G. Hunt and D. Brubacher, "Detours: Binary interception of Win32 functions," in *Proc. 3rd Conf. USENIX Windows NT Symp.*, vol. 3. Seattle, WA, USA, 1999, p. 14.

[35] Protean Security. (2013). *API Hooking and DLL Injection on Windows*. [Online]. Available: http://resources.infosecinstitute.com/api-hooking-and-dll-injection-on-windows

[36] Protean Security. (2013). *Using SetWindowsHookEx for DLL Injection on Windows*. [Online]. Available: http://resources.infosecinstitute.com/using-setwindowshookex-for-dll-injection-on-windows

[37] Protean Security. (2013). *Using CreateRemoteThread for DLL Injection on Windows*. [Online]. Available: http://resources.infosecinstitute.com/using-createremotethread-for-dll-injection-on-windows

[38] Microsoft Developer Network. *SetWindowsHookEx Function (Windows)*. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms644990(v=vs.85).aspx

[39] Microsoft Developer Network. *Hooks Overview (Windows)*. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/ms644959(v=vs.85).aspx

[40] D. Harley. (Oct. 2011). *TDL4 Rebooted*. [Online]. Available: http://www.welivesecurity.com/2011/10/18/tdl4-rebooted/

[41] *Sony's DRM Rootkit: The Real Story*. (2005). [Online]. Available: http://www.schneier.com/blog/archives/2005/11/sonys_drm_rootk.html

[42] P. Ferrie. *The Curse of Necurs—Part 1*. [Online]. Available: http://www.virusbtn.com/pdf/magazine/2014/201404.pdf

[43] H. Li, S. Zhu, and J. Xie. *RTF Attack Takes Advantage of Multiple Exploits*. [Online]. Available: https://blogs.mcafee.com/mcafee-labs/rtf-attack-takes-advantage-of-multiple-exploits/

[44] *Hooking the System Service Dispatch Table (SSDT)—InfoSec Resources*. [Online]. Available: http://resources.infosecinstitute.com/hooking-system-service-dispatch-table-ssdt/

[45] *Hooking IDT—InfoSec Resources*. [Online]. Available: http://resources.infosecinstitute.com/hooking-idt/

[46] A. S. Tanenbaum, *Modern Operating Systems*, 3rd ed. Upper Saddle River, NJ, USA: Prentice-Hall, 2007.

[47] Microsoft Developer Network. *PsSetLoadImageNotifyRoutine Routine (Windows Drivers)*. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/hardware/ff559957(v=vs.85).aspx

[48] B. Jack, "Remote windows kernel exploitation: Step into ring 0," eEye Digit. Security, Aliso Viejo, CA, USA, Tech. Rep., 2005.

[49] A. Srivastava, A. Lanzi, J. Giffin, and D. Balzarotti, "Operating system interface obfuscation and the revealing of hidden operations," in *Proc. 8th Int. Conf. Detection Intrusions Malware Vulnerability Assess.*, Amsterdam, The Netherlands, 2011, pp. 214–233.

[50] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting stealth software with strider Ghostbuster," in *Proc. Int. Conf. Depend. Syst. Netw.*, Yokohama, Japan, 2005, pp. 368–377.

[51] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting Kernel-level rootkits using data structure invariants," *IEEE Trans. Depend. Secure Comput.*, vol. 8, no. 5, pp. 670–684, Sep./Oct. 2011.

[52] K. Seifried, "Fourth-generation rootkits," *Linux Mag.*, no. 97, 2008. [Online]. Available: http://www.linux-magazine.com/Issues/2008/97/Security-Lessons

[53] P. Ször and P. Ferrie, "Hunting for metamorphic," in *Proc. Virus Bull. Conf.*, 2001, pp. 123–144.

[54] P. Beaucamps, "Advanced polymorphic techniques," *Int. J. Comput. Sci.*, vol. 2, no. 3, pp. 194–205, 2007.

[55] S. M. Sridhara and M. Stamp, "Metamorphic worm that carries its own morphing engine," *J. Comput. Virol. Hacking Tech.*, vol. 9, no. 2, pp. 49–58, 2013.

[56] É. Filiol, "Metamorphism, formal grammars and undecidable code mutation," *Int. J. Comput. Sci.*, vol. 2, no. 1, pp. 70–75, 2007.

[57] P. V. Zbitskiy, "Code mutation techniques by means of formal grammars and automatons," *J. Comput. Virol.*, vol. 5, no. 3, pp. 199–207, 2009.

[58] P. Faruki *et al.*, "Evaluation of android anti-malware techniques against Dalvik bytecode obfuscation," in *Proc. IEEE 13th Int. Conf. Trust Security Privacy Comput. Commun.*, Beijing, China, 2014, pp. 414–421.

[59] P. Faruki, S. Bhandari, V. Laxmi, M. Gaur, and M. Conti, "DroidAnalyst: Synergic app framework for static and dynamic app analysis," in *Recent Advances in Computational Intelligence in Defense and Security*. Cham, Switzerland: Springer, 2016, pp. 519–552.

[60] B. Bencsáth, G. Pék, L. Buttyán, and M. Félegyházi, "The cousins of Stuxnet: Duqu, flame, and Gauss," *Future Internet*, vol. 4, no. 4, pp. 971–1003, 2012.

[61] *Apple IDs Targeted by Kelihos Botnet Phishing Campaign*. [Online]. Available: http://www.symantec.com/connect/blogs/apple-ids-targeted-kelihos-botnet-phishing-campaign

[62] *An Encounter With Trojan Nap*, FireEye Threat Res. Blog, Milpitas, CA, USA. [Online]. Available: https://www.fireeye.com/blog/threat-research/2013/02/an-encounter-with-trojan-nap.html

[63] J. T. Bennett, N. Moran, and N. Villeneuve, *Poison Ivy: Assessing Damage and Extracting Intelligence*, Fireeye Threat Res. Blog, Milpitas, CA, USA, 2013. [Online]. Available: http://www.FireEye.com/resources/pdfs/FireEye-poison-ivy-report.pdf

[64] *Trojan Upclicker Ties Malware to the Mouse*, InfoSecurity Mag., Richmond, U.K. [Online]. Available: http://www.infosecurity-magazine.com/news/trojan-upclicker-ties-malware-to-the-mouse/

[65] A. Singh and Z. Bu, *Hot Knives Through Butter: Evading File-Based Sandboxes*, Threat Res. Blog, Milpitas, CA, USA, 2013. [Online]. Available: https://www.fireeye.com/blog/threat-research/2013/08/hot-knives-through-butter-bypassing-file-based-sandboxes.html

[66] *Internet Security Threat Report: Trends for 2016*, Symantec, Mountain View, CA, USA, vol. 21, p. 81, 2016.

[67] T. Micro, "Point-of-sale system breaches: Threats to the retail and hospitality industries," Trend Micro Inc., Tokyo, Japan, Tech. Rep., 2014. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-pos-system-breaches.pdf

[68] *Darkhotel's Attacks in 2015—Securelist*. [Online]. Available: https://securelist.com/blog/research/71713/darkhotels-attacks-in-2015/

[69] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet dossier," White Paper, Symantec Corp., Security Response, vol. 5, p. 6, 2011.

[70] "Gauss: Abnormal distribution," Kaspersky Lab Glob. Res. Anal. Team, Kaspersky Lab, Moscow, Russia, Tech. Rep., 2012.

[71] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi, "sKyWIper (a.k.a Flame a.k.a Flamer): A complex malware for targeted attacks," Dept. Telecommun., Budapest Univ. Technol. Econ., Budapest, Hungary, CrySyS Lab Tech. Rep. CTR-2012-05-31, 2012.

[72] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2003.

[73] J. Rutkowska, "System virginity verifier—Defining the roadmap for malware detection on windows systems," in *Proc. 5th Hack Box Security Conf.*, 2005.

[74] J. Rutkowska. (2004). *Detecting Windows Server Compromises With Patchfinder 2*. [Online]. Available: http://repo.hackerzvoice.net/depot_madchat/vxdevl/avtech/Detecting%20Windows%20Server%20Compromises%20with%20Patchfinder%202.pdf

[75] J. Rutkowska. (2005). *Thoughts About Cross-View Based Rootkit Detection*. [Online]. Available: https://vxheaven.org/lib/pdf/Thoughts%20about%20Cross-View%20based%20Rootkit%20Detection.pdf

[76] J. Butler and S. Sparks, "Windows rootkits of 2005, part three," Symantec Connect Community, Tech. Rep., 2005. [Online]. Available: http://www.symantec.com/connect/articles/windows-rootkits-2005-part-three

[77] B. Cogswell and M. Russinovich, "RootkitRevealer," *Rootkit Detection Tool by Microsoft*, vol. 1, 2006.

[78] J. Butler and P. Silberman, "RAIDE: Rootkit analysis identification elimination," *Black Hat USA*, vol. 47, 2006.

[79] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proc. 15th Conf. USENIX Security Symp.*, vol. 2. Vancouver, BC, Canada, 2006, p. 20.

[80] M. Siddiqui, M. C. Wang, and J. Lee, "A survey of data mining techniques for malware detection using file features," in *Proc. 46th Annu. Southeast Regional Conf.*, Auburn, CA, USA, 2008, pp. 509–510.

[81] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for Kernel data structures," in *Proc. 16th ACM Conf. Comput. Commun. Security*, Chicago, IL, USA, 2009, pp. 566–577.

[82] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 335–350, 2007.

[83] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *Proc. 11th Int. Symp. Recent Adv. Intrusion Detect.*, Boston, MA, USA, 2008, pp. 1–20.

[84] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[85] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, Nuremberg, Germany, 2009, pp. 47–60.

[86] V. Ganapathy, T. Jaeger, and S. Jha, "Automatic placement of authorization hooks in the Linux security modules framework," in *Proc. 12th ACM Conf. Comput. Commun. Security*, Alexandria, VA, USA, 2005, pp. 330–339.

[87] J.-W. Jang, J. Yun, A. Mohaisen, J. Woo, and H. K. Kim, "Detecting and classifying method based on similarity matching of Android malware behavior with profile," *SpringerPlus*, vol. 5, no. 1, p. 1, 2016.

[88] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-gram-based detection of new malicious code," in *Proc. 28th Annu. Int. Comput. Softw. Appl. Conf.*, vol. 2. Hong Kong, 2004, pp. 41–42.

[89] D. K. S. Reddy, S. K. Dash, and A. K. Pujari, "New malicious code detection using variable length *n*-grams," in *Proc. 2nd Int. Conf. Inf. Syst. Security*, Kolkata, India, 2006, pp. 276–288.

[90] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding your garbage: Reducing data lifetime through secure deallocation," in *Proc. 14th Conf. USENIX Security Symp.*, Baltimore, MD, USA, 2005, p. 22.

[91] A. Schuster, "Searching for processes and threads in Microsoft Windows memory dumps," *Digit. Investigation*, vol. 3, no. 1, pp. 10–16, 2006.

[92] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. IEEE Symp. Security Privacy*, Berkeley, CA, USA, 2003, pp. 62–75.

[93] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for UNIX processes," in *Proc. IEEE Symp. Security Privacy*, Oakland, CA, USA, 1996, pp. 120–128.

[94] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *Proc. 11th USENIX Security Symp.*, San Francisco, CA, USA, 2002, pp. 61–79.

[95] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Security*, vol. 6, no. 3, pp. 151–180, 1998.

[96] M. Krohn *et al.*, "Information flow control for standard OS abstractions," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 321–334, 2007.

[97] D. Mutz, W. K. Robertson, G. Vigna, and R. A. Kemmerer, "Exploiting execution context for the detection of anomalous system calls," in *Proc. 10th Int. Symp. Recent Adv. Intrusion Detection*, Gold Coast, QLD, Australia, 2007, pp. 1–20.

[98] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. Symp. Security Privacy*, Oakland, CA, USA, 2001, pp. 144–155.

[99] S. Yu, S. Guo, and I. Stojmenovic, "Fool me if you can: Mimicking attacks and anti-attacks in cyberspace," *IEEE Trans. Comput.*, vol. 64, no. 1, pp. 139–151, Jan. 2015.

[100] S. Yu, G. Wang, and W. Zhou, "Modeling malicious activities in cyber space," *IEEE Netw.*, vol. 29, no. 6, pp. 83–87, Sep./Nov. 2015.

[101] G. A. Fink, P. Muessig, and C. North, "Visual correlation of host processes and network traffic," in *Proc. IEEE Workshop Visualization Comput. Security*, Minneapolis, MN, USA, 2005, pp. 11–19.

[102] C. M. Bishop, *Pattern Recognition and Machine Learning* (Information Science and Statistics). New York, NY, USA: Springer, 2006.

[103] G. O'Gorman and G. McDonald. (2012). *Ransomware: A Growing Menace*. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ransomware-a-growing-menace.pdf

[104] A. Venkatesan, "Code obfuscation and virus detection," M.S. thesis, Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2008.

[105] S. Venkatachalam, "Detecting undetectable computer viruses," M.S. thesis, Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2010.

[106] N. Runwal, R. M. Low, and M. Stamp, "Opcode graph similarity and metamorphic detection," *J. Comput. Virol.*, vol. 8, no. 1, pp. 37–52, 2012.

[107] D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *J. Comput. Virol.*, vol. 7, no. 3, pp. 201–214, 2011.

[108] P. Desai, "Towards an undetectable computer virus," M.S. thesis, Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2008.

[109] W. Wong, "Analysis and detection of metamorphic computer viruses," M.S. thesis, Dept. Comput. Sci., San Jose State Univ., San Jose, CA, USA, 2006.

[110] W. Wong and M. Stamp, "Hunting for metamorphic engines," *J. Comput. Virol.*, vol. 2, no. 3, pp. 211–229, 2006.

[111] S. Attaluri, S. McGhee, and M. Stamp, "Profile hidden Markov models and metamorphic virus detection," *J. Comput. Virol.*, vol. 5, no. 2, pp. 151–169, 2009.

[112] S. Zanero and S. M. Savaresi, "Unsupervised learning techniques for an intrusion detection system," in *Proc. ACM Symp. Appl. Comput.*, Nicosia, Cyprus, 2004, pp. 412–419.

[113] W. J. Scheirer, A. Rocha, A. Sapkota, and T. E. Boult, "Toward open set recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 35, no. 7, pp. 1757–1772, Jul. 2013.

[114] M. M. Masud, J. Gao, L. Khan, J. Han, and B. Thuraisingham, "Classification and novel class detection in concept-drifting data streams under time constraints," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 6, pp. 859–874, Jun. 2011.

[115] S. Mukkamala, G. Janoski, and A. Sung, "Intrusion detection using neural networks and support vector machines," in *Proc. Int. Joint Conf. Neural Netw.*, vol. 2. Honolulu, HI, USA, 2002, pp. 1702–1707.

[116] C. A. Catania and C. G. Garino, "Automatic network intrusion detection: Current techniques and open issues," *Comput. Elect. Eng.*, vol. 38, no. 5, pp. 1062–1072, 2012.

[117] W. Lee and S. J. Stolfo, "Data mining approaches for intrusion detection," in *Proc. 7th Conf. USENIX Security Symp.*, San Antonio, TX, USA, 1998, p. 6.

[118] L. Portnoy, E. Eskin, and S. Stolfo, "Intrusion detection with unlabeled data using clustering," in *Proc. ACM CSS Workshop Data Min. Appl. Security*, Philadelphia, PA, USA, 2001, pp. 5–8.

[119] A. Lazarevic, L. Ertöz, V. Kumar, A. Ozgur, and J. Srivastava, "A comparative study of anomaly detection schemes in network intrusion detection," in *Proc. 3rd SIAM Int. Conf. Data Min.*, San Francisco, CA, USA, 2003, pp. 25–36.

[120] L. Ertöz *et al.*, "The MINDS—Minnesota intrusion detection system," in *Next Generation Data Mining*. MIT Press, 2004, ch. 3.

[121] M. Rehák *et al.*, "Adaptive multiagent system for network traffic monitoring," *IEEE Intell. Syst.*, vol. 24, no. 3, pp. 16–25, May/Jun. 2009.

[122] C. C. Aggarwal, Ed., *Data Streams—Models and Algorithms* (Advances in Database Systems), vol. 31. New York, NY, USA: Springer, 2007.

[123] G. G. Helmer, J. S. K. Wong, V. Honavar, and L. Miller, "Intelligent agents for intrusion detection," in *Proc. IEEE Inf. Technol. Conf.*, Syracuse, NY, USA, 1998, pp. 121–124.

[124] I. Ahmad, A. B. Abdulah, A. S. Alghamdi, K. Alnfajan, and M. Hussain, "Feature subset selection for network intrusion detection mechanism using genetic eigenvectors," in *Proc. Int. Conf. Telecommun. Technol. Appl.*, 2011, pp. 75–79.

[125] H. Nguyen, K. Franke, and S. Petrović, "Improving effectiveness of intrusion detection by correlation feature selection," in *Proc. 5th Int. Conf. Availability Rel. Security*, Kraków, Poland, 2010, pp. 17–24.

[126] S. Lakhina, S. Joseph, and B. Verma, "Feature reduction using principal component analysis for effective anomaly-based intrusion detection on NSL-KDD," *Int. J. Eng. Sci. Technol.*, vol. 2, no. 6, pp. 1790–1799, 2010.

[127] M. Middlemiss and G. Dick, "Feature selection of intrusion detection data using a hybrid genetic algorithm/KNN approach," in *Design and Application of Hybrid Intelligent Systems*, A. Abraham, M. Köppen, and K. Franke, Eds. Amsterdam, The Netherlands: IOS Press, 2003, pp. 519–527.

[128] H.-F. Yu *et al.*, "Feature engineering and classifier ensemble for KDD cup 2010," in *Proc. KDD Cup Workshop*, 2010, pp. 1–16.

[129] S. Mukkamala and A. H. Sung, "Feature selection for intrusion detection with neural networks and support vector machines," *Transp. Res. Rec. J. Transp. Res. Board*, vol. 1822, no. 1, pp. 33–39, 2003.

[130] G. Stein, B. Chen, A. S. Wu, and K. A. Hua, "Decision tree classifier for network intrusion detection with GA-based feature selection," in *Proc. 43rd Annu. Southeast Regional Conf.*, vol. 2. Kennesaw, GA, USA, 2005, pp. 136–141.

[131] A. H. Sung and S. Mukkamala, "Identifying important features for intrusion detection using support vector machines and neural networks," in *Proc. Symp. Appl. Internet*, Orlando, FL, USA, 2003, pp. 209–217.

[132] G. Wang, J. Hao, J. Ma, and L. Huang, "A new approach to intrusion detection using artificial neural networks and fuzzy clustering," *Expert Syst. Appl.*, vol. 37, no. 9, pp. 6225–6232, 2010.

[133] M. Amini, R. Jalili, and H. R. Shahriari, "RT-UNNID: A practical solution to real-time network-based intrusion detection using unsupervised neural networks," *Comput. Security*, vol. 25, no. 6, pp. 459–468, 2006.

[134] G. Liu, Z. Yi, and S. Yang, "Letters: A hierarchical intrusion detection model based on the PCA neural networks," *Neurocomputing*, vol. 70, nos. 7–9, pp. 1561–1568, 2007.

[135] T. Srinivasan, V. Vijaykumar, and R. Chandrasekar, "A self-organized agent-based architecture for power-aware intrusion detection in wireless ad-hoc networks," in *Proc. Int. Conf. Comput. Informat.*, Kuala Lumpur, Malaysia, 2006, pp. 1–6.

[136] J. Shun and H. A. Malki, "Network intrusion detection system using neural networks," in *Proc. 4th Int. Conf. Nat. Comput.*, vol. 5. Jinan, China, 2008, pp. 242–246.

[137] A. M. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Boston, MA, USA, 2015, pp. 427–436.

[138] C. Szegedy *et al.*, "Intriguing properties of neural networks," in *Proc. Int. Conf. Learn. Represent.*, Banff, AB, Canada, 2014.

[139] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *Proc. Int. Conf. Learn. Represent.*, San Diego, CA, USA, 2015.

[140] F. V. Jensen and T. D. Nielsen, *Bayesian Networks and Decision Graphs*, 2nd ed. New York, NY, USA: Springer, 2007.

[141] S. Kotz and S. Nadarajah, *Extreme Value Distributions: Theory and Applications*. London, U.K.: Imperial College Press, 2000.

[142] M. Markou and S. Singh, "Novelty detection: A review—Part 1: Statistical approaches," *Signal Process.*, vol. 83, no. 12, pp. 2481–2497, 2003.

[143] W. J. Scheirer, L. P. Jain, and T. E. Boult, "Probability models for open set recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 36, no. 11, pp. 2317–2324, Nov. 2014.

[144] L. P. Jain, W. J. Scheirer, and T. E. Boult, "Multi-class open set recognition using probability of inclusion," in *Proc. Eur. Conf. Comput. Vis.*, Zürich, Switzerland, 2014, pp. 385–388.

[145] A. Bendale and T. E. Boult, "Towards open world recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Boston, MA, USA, 2015, pp. 1893–1902.

[146] D.-Y. Yeung and C. Chow, "Parzen-window network intrusion detectors," in *Proc. 16th Int. Conf. Pattern Recognit.*, vol. 4. Quebec City, QC, Canada, 2002, pp. 385–388.

[147] H. Alizadeh, A. Khoshrou, and A. Zúquete, "Traffic classification and verification using unsupervised learning of Gaussian mixture models," in *Proc. Int. Workshop Measur. Netw.*, Coimbra, Portugal, 2015, pp. 1–6.

[148] W. Fan, N. Bouguila, and H. Sallay, "Anomaly intrusion detection using incremental learning of an infinite mixture model with feature selection," in *Proc. 8th Int. Conf. Rough Sets Knowl. Technol.*, Halifax, NS, Canada, 2013, pp. 364–373.

[149] C. Gruhl, B. Sick, A. Wacker, S. Tomforde, and J. Hähner, "A building block for awareness in technical systems: Online novelty detection and reaction with an application in intrusion detection," in *Proc. 7th Int. Conf. Awareness Sci. Technol.*, Qinhuangdao, China, 2015, pp. 194–200.

[150] P. Lam, L.-L. Wang, H. Y. T. Ngan, N. H. C. Yung, and A. G.-O. Yeh, "Outlier detection in large-scale traffic data by Naïve Bayes method and Gaussian mixture model method," arXiv preprint, 2015. [Online]. Available: http://arxiv.org/abs/1512.08413

[151] K. Yamanishi, J.-I. Takeuchi, G. Williams, and P. Milne, "On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms," *Data Min. Knowl. Disc.*, vol. 8, no. 3, pp. 275–300, 2004.

[152] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognit.*, vol. 36, no. 1, pp. 229–243, 2003.

[153] M. Amer, M. Goldstein, and S. Abdennadher, "Enhancing one-class support vector machines for unsupervised anomaly detection," in *Proc. ACM SIGKDD Workshop Outlier Detection Description*, Chicago, IL, USA, 2013, pp. 8–15.

[154] J. Yang, T. Deng, and R. Sui, "An adaptive weighted one-class SVM for robust outlier detection," in *Proceedings of the 2015 Chinese Intelligent Systems Conference*. Heidelberg, Germany: Springer, 2015, pp. 475–484.

[155] K. A. Heller, K. M. Svore, A. D. Keromytis, and S. J. Stolfo, "One class support vector machines for detecting anomalous windows registry accesses," in *Proc. Workshop Data Min. Comput. Security*, 2003.

[156] Y. Wang, J. Wong, and A. Miner, "Anomaly intrusion detection using one-class SVM," in *Proc. 5th Annu. IEEE SMC Inf. Assurance Workshop*, West Point, NY, USA, 2004, pp. 358–364.

[157] K.-L. Li, H.-K. Huang, S.-F. Tian, and W. Xu, "Improving one-class SVM for anomaly detection," in *Proc. Int. Conf. Mach. Learn. Cybern.*, vol. 5. Xi'an, China, 2003, pp. 3077–3081.

[158] R. Perdisci, G. Gu, and W. Lee, "Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems," in *Proc. 6th Int. Conf. Data Min.*, Hong Kong, 2006, pp. 488–498.
[159] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson, "Estimating the support of a high-dimensional distribution," *Neural Comput.*, vol. 13, no. 7, pp. 1443–1471, 2001.
[160] E. M. Rudd, L. P. Jain, W. J. Scheirer, and T. E. Boult, "The extreme value machine," arXiv preprint, 2015. [Online]. Available: http://arxiv.org/abs/1506.06112
[161] T. Lane and C. E. Brodley, "Approaches to online learning and concept drift for user identification in computer security," in *Proc. 4th Int. Conf. Knowl. Disc. Data Min.*, New York, NY, USA, 1998, pp. 259–263.
[162] R. R. Karthick, V. P. Hattiwale, and B. Ravindran, "Adaptive network intrusion detection system using a hybrid approach," in *Proc. 4th Int. Conf. Commun. Syst. Netw.*, Bengaluru, India, 2012, pp. 1–7.
[163] K. Wang and S. J. Stolfo, "Anomalous payload-based network intrusion detection," in *Proc. 7th Int. Symp. Recent Adv. Intrusion Detect.*, Sophia Antipolis, France, 2004, pp. 203–222.
[164] S. Zhong, T. M. Khoshgoftaar, and N. Seliya, "Clustering-based network intrusion detection," *Int. J. Rel. Qual. Safety Eng.*, vol. 14, no. 2, pp. 169–187, 2007.
[165] J. Cannady, "Applying CMAC-based online learning to intrusion detection," in *Proc. IEEE-INNS-ENNS Int. Joint Conf. Neural Netw.*, vol. 5. Como, Italy, 2000, pp. 405–410.
[166] W. Hu, J. Gao, Y. Wang, O. Wu, and S. Maybank, "Online Adaboost-Based parameterized methods for dynamic distributed network intrusion detection," *IEEE Trans. Cybern.*, vol. 44, no. 1, pp. 66–82, Jan. 2014.
[167] S. Wang *et al.*, "Concept drift detection for online class imbalance learning," in *Proc. IEEE Int. Joint Conf. Neural Netw.*, Dallas, TX, USA, 2013, pp. 1–10.
[168] W. J. Scheirer, A. Rocha, R. J. Michaels, and T. E. Boult, "Robust fusion: Extreme value theory for recognition score normalization," in *Proc. 11th Eur. Conf. Comput. Vis.*, Heraklion, Greece, 2010, pp. 481–495.
[169] W. J. Scheirer, N. Kumar, P. N. Belhumeur, and T. E. Boult, "Multi-attribute spaces: Calibration for attribute fusion and similarity search," in *Proc. 25th IEEE Conf. Comput. Vis. Pattern Recognit.*, Providence, RI, USA, 2012, pp. 2933–2940.
[170] G. Creech and J. Hu, "A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 807–819, Apr. 2014.
[171] G. Creech and J. Hu, "Generation of a new IDS test dataset: Time to retire the KDD collection," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Shanghai, China, 2013, pp. 4487–4492.
[172] G. Creech, "Developing a high-accuracy cross platform host-based intrusion detection system capable of reliably detecting zero-day attacks," Ph.D. dissertation, Dept. Eng. Inf. Technol., Univ. New South Wales, Sydney, NSW, Australia, 2014.
[173] Microsoft Developer Network. *API Index*. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/hh920508(v=vs.85).aspx
[174] Wikipedia. *Microsoft Windows Library Files*. [Online]. Available: https://en.wikipedia.org/wiki/Microsoft_Windows_library_files

Ethan M. Rudd received the B.S. degree in physics from Trinity University in 2012, and the M.S. degree in computer science from the University of Colorado at Colorado Springs in 2014, where he is currently pursuing the Ph.D. degree in computer science. He was with the Vision and Security Technology Laboratory, University of Colorado at Colorado Springs, conducting research in computer vision, biometrics, and applied machine learning.

Andras Rozsa received the M.Eng. degree in information technology from the University of Veszprem, Hungary, in 2005, and the M.S. degree in computer science from the University of Colorado at Colorado Springs (UCCS) in 2014, where he is currently pursuing the Ph.D. degree in engineering with a specialty in security. He was with the Vision and Security Technology Laboratory, UCCS, researching machine learning and deep neural network applications to computer vision.

Manuel Günther received the diploma degree in computer science with a major subject of machine learning from the Technical University of Ilmenau, Germany, in 2004, and the doctoral degree from the Ruhr University of Bochum, Germany, under the thesis entitled "Statistical Extensions of Gabor Graph Based Face Detection, Recognition and Classification Techniques," in 2011. From 2012 to 2015, he was a Post-Doctoral Researcher with the Biometrics Group, Idiap Research Institute, Martigny, Switzerland, where he was actively participating in the implementation of the open source signal processing and machine learning library Bob, particularly he was the Leading Developer of the bob.bio packages. Since 2015, he has been a Research Associate with the Vision and Security Technology Laboratory, University of Colorado at Colorado Springs. He is currently a part of the IARPA Janus Research Team and is occupied with incorporating facial attributes to build more reliable face recognition algorithms in uncontrolled imaging environments.

Terrance E. Boult was an Endowed Professor and the Founding Chairman with the CSE Department, Lehigh University, and from 1986 to 1992, he was a faculty member with Columbia University. In 2003, he joined the University of Colorado at Colorado Springs (UCCS), where he is an El Pomar Professor of Innovation and Security researching on computer vision, machine learning, biometrics, and security. He is an innovator with a passion for combining teaching, research, and business. He was a recipient of multiple teaching, research, innovation, and entrepreneurial awards. At UCCS, he was the architect of the awarding winning Bachelor of Innovation family of degrees and a key member in founding the UCCS Ph.D. in engineering security. He has been involved with multiple start up companies in the security space.