Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds

Antonio Bianchi, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna
UC Santa Barbara
Santa Barbara, CA, USA
{antoniob,yans,chris,vigna}@cs.ucsb.edu

ABSTRACT

The lucrative rewards of security penetrations into large organizations have motivated the development and use of many sophisticated rootkit techniques to maintain an attacker's presence on a compromised system. Due to the evasive nature of such infections, detecting these rootkit infestations is a problem facing modern organizations. While many approaches to this problem have been proposed, various drawbacks that range from signature generation issues, to coverage, to performance, prevent these approaches from being ideal solutions.

In this paper, we present *Blacksheep*, a distributed system for detecting a rootkit infestation among groups of similar machines. This approach was motivated by the homogenous natures of many corporate networks. Taking advantage of the similarity amongst the machines that it analyses, *Blacksheep* is able to efficiently and effectively detect both existing and new infestations by comparing the memory dumps collected from each host.

We evaluate *Blacksheep* on two sets of memory dumps. One set is taken from virtual machines using virtual machine introspection, mimicking the deployment of *Blacksheep* on a cloud computing provider's network. The other set is taken from Windows XP machines via a memory acquisition driver, demonstrating *Blacksheep*'s usage under more challenging image acquisition conditions. The results of the evaluation show that by leveraging the homogeneous nature of groups of computers, it is possible to detect rootkit infestations.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

Keywords

computer security, rootkit detection, kernel-based rootkits

1. INTRODUCTION

Over the past several years, computer security has taken the center stage, as several high-profile organizations have suffered costly intrusions. Oftentimes, as in the case of the 2011 RSA compromise, such intrusions begin as a foothold on a single infected machine and spread out from that foothold to infect a larger portion of the enterprise. In the case of the 2010 Stuxnet attack on Irani nuclear reactors, this infection took the form of a kernel-based rootkit.

Rootkits are pieces of software designed to stealthily modify the behavior of an operating system in order to achieve malicious goals, such as hiding user space objects (e.g., processes, files, and network connections), logging user keystrokes, disabling security software, and installing backdoors for persistent access. Although several detection and prevention techniques have been developed and deployed, all have considerable drawbacks, and as a result, rootkits remain a security threat: according to recent estimates, the percentage of rootkits among all anti-virus detections is in the range of 7-10% [17, 32].

The situation is further complicated by the fact that rootkit evasion techniques are continuously evolving [17]. One recent development that has greatly complicated rootkit detection is the emergence of rootkits that work solely by modifying data, rendering tools that focus on detecting code changes (such as the System Virginity Verifier [27]) ineffective. This drawback applies to most current detection techniques, rendering them ineffective against memory-only rootkits.

The goal of our work is to detect kernel rootkits, a broad class of rootkits that operate by modifying kernel code or kernel data structures. We focus on the Windows operating system, since it is both the most widespread and the most targeted platform. However, most of the concepts and techniques used are applicable to any operating system.

The observation that motivates our approach to the detection of rootkits is the fact that modern organizations rely on large networks of computers to accomplish their daily workflows. In order to simplify maintenance, upgrades, and replacement of their computers, organizations tend to utilize a standard set of software and settings for the configuration of these machines. For example, a large company might make a standard image for employee workstations, another image for servers, a third image for virtualized deployments, and so forth. At the same time, such nearly-identical computers are treated as unique entities when enforcing security policies and scanning for malware. We believe that by leveraging the similarities between these computers, rootkits can be detected with higher accuracy and without the limitations of modern rootkit detection techniques.

Therefore, we propose a novel technique for detecting kernel rootkits, based on the analysis of physical memory dumps taken from running operating systems. In our approach, a set of memory dumps from a population of computers with identical (or similar) hardware and software configurations are taken. These dumps are then compared with each other to find groups of machines that are similar. Finally, these groups are further analyzed to identify the kernel modifications introduced by a potential rootkit infection. In particular, we look for outliers that are different than the rest. Our insight is that these differences are an indication of a malware infection.

We implemented our approach in a tool, called *Blacksheep*, and validated it by analyzing memory dumps taken from two sets of computers. From each set, *Blacksheep* is able to detect kernel modifications introduced by all the kernel rootkits that we tested and can successfully discriminate between memory dumps taken from non-infected and infected computers.

Blacksheep has several advantages over the state of the art. First of all, *Blacksheep* can detect stealthy rootkit infection techniques, such as data-only modifications of kernel memory. Additionally, *Blacksheep* does not need to be configured to detect specific modifications, because it relies on the identification of anomalies among a group of similar hosts. This means that *Blacksheep* does not use or rely on signatures, and can detect 0-days as effectively as it can detect long-known threats.

Since *Blacksheep* bases its analysis off of a crowd of similarly-configured machines, the system can be used on groups of machines in which some instances are already infected with malware. As long as a viable memory dump can be obtained, and as long as the majority of the machines comprising the crowd are not compromised, *Blacksheep* will be able to identify infections by comparing the memory dumps of the involved machines. In contrast, prior tools that utilize comparative techniques on data from a single machine cannot be safely deployed onto infected computers, since they would then have no safe baseline against which to compare.

Finally, because *Blacksheep* detects the differences among the computers in a crowd, anti-virus software that modifies the kernel (often producing false positives for other rootkit detection techniques) can be properly accommodated, as such software would be deployed on all machines. Unstable sections of the Windows kernel, such as pages that contain self-modifying code for security purposes, can also be handled, since such sections will differ on each member of the crowd, and *Blacksheep* will not regard the differences as suspicious.

Note that while we have implemented *Blacksheep* for Windows XP and Windows 7, the approach that we take can be generalized to any operating system with kernel memory.
In summary, our contributions are:

1. **Forensics**: We detail our forensic investigation into the Windows kernel and describe the considerations that must be taken into account to successfully compare memory dumps from two machines and to obtain a meaningful similarity measure.

2. **Detection**: We present and implement an approach utilizing memory similarities to detect anomalies in a group of similar machines. Our approach can detect rootkits that use stealthy techniques to evade detection.

As part of our investigation into Windows kernel rootkits, much research needed to be done on the internals of the Windows kernel itself. Part of our contribution is the summary of this research, hoping that it will be useful to other researchers.

The rest of this paper is structured as follows. Section 2 details the prior work in the field. Section 3 covers a high-level overview of our approach. Section 4 covers the technical details of our implementation. We present our evaluation in Section 5, and a discussion of our system and its limitations in Section 6. Finally, we conclude the paper in Section 7.

2. RELATED WORK

A considerable amount of research has been done towards detecting and defending against rootkit infections. In this section, we will discuss the state of the art and show where and how *Blacksheep* improves on such approaches.

2.1 Signature-based detection

The traditional method to detect malware is to match a suspected piece of malware against a database of byte-level signatures describing invariant content of known malicious software [14, 20]. Although this technique is still widely used, it suffers from several major limitations. To begin with, the number of signatures that are required to detect currently known malware infections is exponentially increasing. Even taking into account only kernel-based malware, it is still difficult to generate signatures that describe polymorphic software. Additionally, writing such signatures takes time, and a completely new piece of malware often enjoys precious unhindered time while new signatures for it are manually generated.

Furthermore, signature-based approaches generally utilize hooks in order to scan software as it is written to disk or loaded for execution. This is often accomplished by hooking system calls and other kernel entry points; however, these methods can be evaded by adequately sophisticated software. For instance, some malware programs avoid saving themselves using the filesystem API and instead write themselves to disk by accessing it directly. Other malware samples utilize undocumented and unmonitored mechanisms to execute themselves, thus evading detection by signature-based antivirus software.

Because *Blacksheep* functions by detecting anomalous memory dumps collected from a group of machines instead of looking for specific signatures of infection, it does not require the use of signatures. As such, it is well-built to handle previously-unseen malware threats.

2.2 Behavioral heuristic analysis

To overcome the limitations of signature-based detection, antivirus software often combines signatures with heuristic behavioral analysis [16]. With this approach, a process is analyzed during runtime and its behavior is monitored for signs of maliciousness. For instance, a process that calls some particular security-critical system calls with certain parameters (e.g., modifying file access permissions or adding boot entries) might be classified as suspicious, and the responsible process might be halted.

Behavior-based analyses are very hard to execute properly. Any framework performing this analysis must have a very good understanding of the direct and indirect effects of monitored events. Such understanding is often imperfect, allowing malware to evade detection, by performing "mimicry attacks", similar to the ones described in [33]. *Blacksheep*'s approach is based on the analysis of the memory footprint of malware as opposed to its behavior, and, therefore, such concerns do not apply.

2.3 Sandbox execution

Certain malware detection schemes execute programs in a virtual environment, isolated from the rest of the operating system, and log the actions performed, looking for the side-effects of an infection, such as the creation of files, or the modification of the registry. This method is a good solution to polymorphic malware, since it does not depend on a signature of the file being analyzed. It can also be combined with behavioral heuristics for a more in-depth analysis.

The two biggest drawbacks of this detection method are performance and evasion. Such systems must wait until the sandbox execution has yielded a classification before starting the program on an actual system. This causes a noticeable delay in startup, affecting the perceived performance of the system. Additionally, many techniques allow malware to detect the presence of an emulated environment and, if one is detected, to modify its execution flow.

Blacksheep does not rely on sandbox execution, but rather examines the modifications that the malware does to kernel memory. As such, the startup speed of applications is not relevant, and evasion is considerably more difficult.

2.4 System integrity checking

In the process of subverting normal system behavior, rootkits must modify critical system code and/or data structures [15]. For this reason, one method for detecting rootkits is the checking of the critical components of an operation system to ensure that they are in an expected state.

64-bit versions of the Windows kernel implement a feature called *Kernel Patch Protection (KPP)* [24]. KPP comprises an obfuscated kernel function that is periodically executed to check the integrity of critical components. [26, Chapter 3.14] contains further information as to which kernel components are checked by KPP.

A similar approach has been implemented in the System Virginity Verifier [27]. This tool is based on the idea that, excluding some specific locations (e.g., relocated pointers, data sections), the image in memory of a kernel module should be equal to the content of the file from which it is loaded.

Other approaches, specifically designed as a defense against function pointer hijacking in kernel memory, have also been developed [35, 37].

Yet more solutions have been proposed that are based on the hardware virtualization features in modern processors [23, 31]. The idea behind these approaches is to take advantage of hardware virtualization to perform integrity verification at a higher privilege level than the one at which the kernel code (and the rootkit) are executed.

One fundamental challenge with these systems is the fact that they must identify a baseline with which to compare the current state of the system that they are protecting. In the case of the System Virginity Verifier, the baseline is defined to be the actual files on disk from which the kernel is loaded. However, malware that is motivated enough could also modify these files, thus corrupting the baseline. In other cases, the state of the system when the software was loaded is used. If the system is already infected when such software is loaded, however, this can also provide an improper baseline. *Blacksheep*'s contribution over these existing systems is the fact that a baseline does not have to be defined. Working on the intuition that a malware infestation begins on a subset of machines, *Blacksheep* can determine a baseline that is unrelated to the integrity of individual machines. Additionally, while most integrity checkers analyze the code of a system, *Blacksheep* also carries out a data analysis. This allows *Blacksheep* to detect rootkits that do not analyze code.

2.5 Cross-view detection

Cross-view detection is another popular rootkit detection technique that is implemented by several detection tools [1, 6, 9]. This approach relies on the fact that the same information about the state of a system can be obtained in different ways. For instance, the presence of a file is commonly detected by utilizing user-level APIs. The information returned by such APIs can be easily altered by a rootkit to hide the presence of files. However, scanning file systems using low-level primitives can often reveal a file hidden by a rootkit. Comparing several sets of similar information obtained by different means can often bring such inconsistencies to light, and reveal the presence of a rootkit.

This approach can be undertaken not only with hidden files, but with unlinked processes, network connections, and other such system artifacts. Unfortunately, the number of such possible intersection points is very large, and the checks must, in general, be developed manually. Thus, missing a modification done by a rootkit is very likely. Since *Blacksheep* examines the entire contents of kernel memory, it does not suffer from this requirement for manual test development.

2.6 Invariant-based detection

The problem of kernel integrity verification is similar to the problem of discovering and verifying invariant properties within kernel memory. In particular, research has been conducted into the detection of such invariants in kernel data structures and the subsequent verification of kernel memory to ensure that it has not been violated by a rootkit.

Petroni et al. propose an architecture to manually specify kernel data invariants and to check them automatically [25]. This architecture allows one to easily declare properties that must hold inside an uncompromised machine. However, manually specifying such properties requires a deep knowledge of operating system internals, and it is particularly difficult when no source code is available. Even if such source code is present, the size of modern operating systems makes manually specifying such invariants extremely difficult. Additionally, even if source code were present and invariants are automatically specified, the ability to load kernel-resident drivers in modern operating systems makes this task impossible, as the contents of the kernel cannot always be known ahead of time with complete certainty. *Blacksheep* does not require such knowledge, and will function as long as the kernel modules in question are present in a sizeable part of the machine crowd.

Other invariant-enforcing frameworks are Hello RootKitty [12] and HyperForce [11]. However, these systems rely on a predetermined list of invariants. *Blacksheep* has no such requirement.

The state of the art in automatic invariants detection is based on Daikon [10]. Daikon is a tool developed to automatically discover pre-conditions and post-conditions that hold when program functions are called. Baliga et al. have adapted Daikon to work on kernel data structures. Their tool, Gibraltar [5], is able to detect previously-known rootkit that modify the data structures of the Linux kernel. However, this tool, and a similar approach implemented for Windows, called KOP [8], requires kernel source code to extract a graph of kernel data structure relationships. Such source code is often unavailable, especially in the case of external drivers. Furthermore, security systems that contain kernel-based components introduce additional complexity into the real-world use of such programs. In contrast, *Blacksheep* requires no knowledge of source code.

The use of invariants based on graph signatures has been implemented by SigGraph [22]. However, SigGraph requires the availability of source code or debug information, while *Blacksheep* has no such requirement. Additionally, similar ideas have been applied to filesystem changes with Seurat [36].

2.7 Physical memory analysis

Physical memory analysis is an active area of research whose aim is to capture reliable and complete information from a live acquisition of the physical memory of a running system. It has been studied mainly in the context of forensic and malware analysis [7,13], and several specialized tools have been developed to perform such analyses. Such tools include HBGary Responder Pro [2] and Volatility [34].

Volatility is an open-source framework for physical memory analysis, containing an extensible plugin structure that allows for the implementation of various analyses. Various plugins have been developed, including those that perform malware detection and analysis [21].

The detection of memory allocated inside the Windows kernel heap has also been studied [29,30], as has the use of information extracted from the Windows swap file [19] (albeit, mainly for forensic purposes).

Blacksheep utilizes Volatility with build-in and specifically developed plugins to support its operation. In addition, it is able to deal with physical memory dumps created by several common tools for memory acquisition.

3. APPROACH

Blacksheep is designed to detect rootkit infestations in kernel memory. *Blacksheep*'s design is motivated by the realization that, regardless of how much a rootkit tries to hide itself, it must still be accessible by the operating system in order to be executed. This concept is known as the Rootkit Paradox [18]. Additionally, even if a rootkit manages to hide its code from the operating system, the data modifications it makes can still be detected. While some conceptual rootkits have been demonstrated that can completely unplug themselves from the system [22], they do so by mangling pointers and destabilizing the victim operating system. Even if the operating system survives these modifications, the pointer manipulations can still be observed.

With this basic idea in mind, we created *Blacksheep*, which compares images of physical memory taken from similar machines to identify differences associated with rootkit infections. *Blacksheep* is most effective when operating on a crowd of similar machines.

Since we are comparing kernel memory snapshots, an understanding of this memory space is required. The Windows kernel consists of many *modules*, which are PE files containing kernel code and data. Modules can be operating system components (e.g., kernel32.dll) or hardware drivers (e.g., nvstor32.sys), and we use these terms interchangeably. The module and driver files are loaded into kernel memory in much the same way as dynamically linked libraries (DLLs) are loaded into user-space programs, and make up the functionality of the kernel. Similar to Windows DLLs, kernel modules contain both code and data segments. These segments require separate approaches in their comparisons, and *Blacksheep* treats them separately.

In summary, *Blacksheep* performs the following four types of analyses, which are detailed in Section 4:

- Configuration comparison;
- Code comparison;
- Data comparison; and
- Kernel entry point comparison.

Configuration comparison. Some rootkits come in the form of a kernel module that is loaded into the system. To identify such changes, *Blacksheep* does a "configuration comparison," comparing loaded modules between two memory dumps. This allows the system to detect additional (and potential malicious) components that are introduced into the kernel. Details are presented in Section 4.1.

Code comparison. Most rootkits directly overwrite or augment existing code in kernel space with malicious content so that they can perform subversive tasks (such as hiding resources) when this code is executed. Thus, a difference in kernel code between machines that should be otherwise identical can be a good indicator that a rootkit may be present. However, due to the possibility of benign differences resulting from, among other causes, code relocation and anti-virus defense techniques, a detected difference might not necessarily mean that the machine is infected. *Blacksheep* can filter out benign differences and focus on suspicious code differences. We discuss the specifics of this functionality in more detail in Section 4.2.

Memory comparison. Detecting differences in kernel code alone is not enough to detect the presence of rootkits with high accuracy. For example, certain rootkits are able to subvert system functionality without performing any modifications to code running on the system, and, instead, they change kernel data structures to avoid detection through code comparison. Because of the threat of such rootkits, we compare kernel data between machines.

Comparing such data between two different machines is a nontrivial task, and constitutes a large portion of *Blacksheep*'s contribution. For statically allocated data segments (i.e., those segments that are defined in and loaded from the PE file), the main challenge is handling relocation. However, dynamically allocated memory provides a more substantial challenge. This data oftentimes contains many layers of data structures linking to each other, which must be navigated in order to ensure good coverage. *Blacksheep* uses several methods to be able to identify and compare such data structures, which are described further in Section 4.3.

Entry point comparison. Additionally, rootkits might subvert basic interfaces to the Windows kernel in order to carry out their tasks. This includes the Windows kernel SSDT, driver IRP communication channels, and certain hardware registers in the x86 architecture. *Blacksheep* is able to compare such *kernel entry points* by processing the machines' dumps of physical memory. We present the details of this kind of analysis in Section 4.4.

Clustering and detection. After comparing each pair of memory dumps, *Blacksheep* places them into clusters, according to the differences present between them. The larger clusters are then assumed to contain the clean dumps, and the smaller clusters are labeled as suspicious. The assumption is that only a small fraction of the hosts are infected, and these hosts stand out as outliers when compared to the other machines in the crowd. This step is discussed in Section 4.5.

4. SYSTEM DETAILS

Blacksheep computes the differences between two memory dumps to produce a *distance metric*. In the computation of differences, *Blacksheep* looks for four categories of differences: high-level configuration differences, code differences, data differences, and differences in kernel entry points.

4.1 Configuration Analysis

Blacksheep is able to utilize configuration information obtained from memory dumps to assist its analysis. Specifically, it is impossible to meaningfully compare the code of a kernel module between two memory dumps if one of the dumps does not have such a driver loaded while the other does. Since rootkits often cause such differences (because they load additional components), *Blacksheep* carries out the comparison of loaded kernel modules as a separate analysis. To accomplish this, a list of loaded kernel modules is identified in each memory dump. Each kernel module is represented as a pair, consisting of the size of it originating PE files and the CRC checksum. The lists are sorted and compared. The distance metric that *Blacksheep* generates for this analysis is equal to the number of differences between the lists of kernel modules. That is, each addition or deletion of a kernel module adds one to the distance value.

Note that some rootkits can (and do) masquerade the modules they inject as common Windows kernel modules. In such an event, the configuration analysis might not find the difference between a malicious driver and a legitimate module installed in another machine. However, this difference will be detected in the subsequent code analysis step instead (as the rootkit code will be very different from the legitimate driver).

4.2 Code Analysis

Most Windows rootkits inject code into kernel memory and redirect legitimate flow of execution to it. *Blacksheep*'s code analysis checks for signs of such redirections by identifying differences in driver code. Since the header information from the PE files of kernel modules is stored in memory, *Blacksheep* can examine the headers of all loaded drivers to identify segments containing driver code.

For each kernel module that is loaded in both memory dumps, *Blacksheep* compares all code segments within both modules, byte-by-byte, to identify a list of bytes that differ. In principle, one could expect that the code segments associated with two identical modules are the same between machines (after all, it is the same code on disk). However, this is not the case, and there are several instances of expected differences that will be present between code segments. *Blacksheep* handles these cases specifically. More precisely, when differing bytes are identified, *Blacksheep* checks them against the following categories, which we consider benign:

Relocation differences. The most frequent differences between driver code segments are caused by relocation. That is, since drivers are loaded into a location in memory that is unknown at compile time, and Windows module code is not position independent, pointers within the driver have to be updated to reflect this location. Since, other than relocation, the relative memory layout of loaded modules is kept intact, relocation differences between two memory dumps can be easily identified. The reason is that the differing bytes will be part of pointers that point to the same *relative* locations within each driver. Thus, when *Blacksheep* finds differing bytes, it first checks whether these bytes are possible pointers (values that point into code segments of the driver). If two such pointers are identified, and they point to the same relative offset into the same driver, this code difference is marked as benign.

Note that on the x86 architecture, pointer locations need not be aligned at word boundaries. Hence, if there is a one-byte difference between the code of two modules in two different memory dumps on a 32-bit system, *Blacksheep* would make four comparisons: one with that byte as the most significant in the pointer, one with it as the second-most significant, and so forth.

Imports and exports. Another benign difference between modules can be caused by imported and exported symbols. These exported symbols take the form of lists of resource names and memory locations. When the drivers that are exporting these symbols are relocated, the export tables are updated accordingly. These benign changes can be detected in a similar way to the detection of relocation differences. If an identified difference is not part of a relocation difference, *Blacksheep* checks if the different bytes are part of a pointer which points to the same offset within some other driver. If the bytes in both dumps are pointing to the same relative offset within the same driver, the difference is considered to be benign.

Hooking. The hooking of kernel functions is another potential source of benign differences. Function call hooking is a technique in which calls to a kernel function are redirected (and cause the execution of some other piece of code). In many cases, this is done by overwriting the first instruction of a hooked function with a jump instruction pointing to the hooking function (so that a call to the hooked function will immediately result in the hooking function being executed). When the hooking function in both memory dumps is located at the same offset in some module (in a static code region), the hook is treated as benign.

Hooks that point to dynamically allocated memory must be treated differently, since their offset to the hooking driver will not be constant. To this end, *Blacksheep* first identifies the hook target: The differing bytes are checked to determine if they are the argument of a *jmp* or *call* instruction. If so, *Blacksheep* calculates the memory addresses pointed to by the hook in the two dumps from the jump target. If the bytes are not used as the argument for a direct control transfer instruction, we check whether they are the argument of a *push* instruction, and if a *ret* instruction follows. The result of executing these instructions would also be a jump to the pushed memory address. In this case, *Blacksheep* recognizes the argument of the push instruction as the location of the hooking function.

Once *Blacksheep* identifies the locations the hooking functions, it needs to compare the functions themselves to detect differences. *Blacksheep* identifies the end of the functions by linearly disassembling them until a *ret* instruction is found. Each byte before the *ret* is then compared using the same mechanism as for regular code segments. Using this method, *Blacksheep* can compare hooks pointing to dynamically allocated memory, which are often used by security software.

PE header differences. Windows sometimes modifies specific fields in the PE header of kernel modules as the modules are loaded. Because of this, we consider differences in the following PE header fields benign:

- ImageBaseAddress
- PointerToRelocations (for each PE section)
- NumberOfRelocations (for each PE section)

Suspicious differences. Any differences that are not classified as benign, according to the above categories, are considered suspicious. Since a common modification done by rootkits is a pointer modification, we count any adjacent set of 4 (or fewer) bytes into one difference. *Blacksheep* uses the number of such differences as the distance metric for its code analysis. Due to the number of changes introduced by rootkit infections, this distance is higher between an infected and clean memory dump than between two clean dumps (or two dumps infected with the same rootkit). In particular, the number of suspicious code differences between two non-infected dumps is usually zero.

4.3 Data Analysis

Recently, proof-of-concept rootkits have been demonstrated that affect the functionality of a system without making any lasting modifications to system code. In order to detect such rootkits, *Blacksheep* must be able to compare kernel data in a sophisticated matter. Windows kernel modules can allocate memory in two different ways: by statically reserving it, as in the various PE data segments, and through dynamic allocation.

To compare data memory between two memory dumps, *Blacksheep* utilizes a “memory crawling” approach to compare kernel data. Memory crawling works as follows: The system processes one memory region at a time, starting at the statically-allocated data regions of each driver. These serve as the roots for the memory exploration. When *Blacksheep* finds pointers to additional data regions (potentially allocated dynamically), it follows these pointers and continues the exploration recursively.

For each region, *Blacksheep* examines the value contained in every (32-bit) dword and assigns it a category, as follows:

ZERO when the dword value == 0x00000000.

VALUE when the dword is a value that does not correspond to a mapped location in memory.

POINTER when the dword is a pointer to a mapped memory location.

Additionally, *Blacksheep* tracks the target of the pointers. For each dword classified as a *POINTER*, *Blacksheep* assigns one of the following subcategories.

POINTER_SELF when the dword is a pointer to the memory location of that pointer (self).

POINTER_NEXT when the dword is a pointer to the dword following itself.

POINTER_CODE when the dword is a pointer into a module's code segment.

POINTER_DATA when the dword is a pointer into a module's data segment.

POINTER_POOL when the dword is a pointer into a dynamically-allocated pool.

POINTER_DLIST when the dword is a pointer to an element in a doubly-linked list.

When *Blacksheep* encounters a *POINTER_POOL* (dynamically-allocated memory) or *POINTER_DLIST* (doubly-linked list) subcategory, additional work is necessary, as discussed in the following two paragraphs.

Dynamically-allocated memory. Dynamic allocation is handled in Windows through the use of memory pools. Each discrete allocation that is requested by a driver is tagged with a pool allocation structure, containing the length of the allocation and a 4-character tag identifying the allocating driver. Before comparing data, *Blacksheep* builds a list of allocated pools inside the kernel. This list can then be used to detect pointers into dynamic memory.

If a dword is of subcategory *POINTER_POOL*, all dwords from the target allocation pool are also added to the analysis. That is, when *Blacksheep* finds a pointer into a pool of memory, this pool is recursively added and analyzed (since it can be reached through a root). Any further dwords that are classified as *POINTER_POOL* are processed recursively. This is done up to three levels of nesting, as a compromise between data coverage, execution time, and noise in the analysis.

Doubly-linked lists. The Windows kernel contains many data structures, and their definitions are available both through the Windows Research Kernel [4] and through Windows debugging symbols. Relying on the availability of such definitions does not work in the general case, however, as many kernel modules (for example, modules from third-party providers such as hardware manufacturers or security companies) do not provide this information. Nevertheless, one extremely common structure is the standard doubly-linked list. This structure is extensively used for various purposes in the Windows kernel, and is easily recognizable in memory due to the layout of the list pointers: the list comprises identically-sized doubly-linked list elements starting and ending at a standard list header. Thus, *Blacksheep* detects and treats doubly-linked lists in a distinct way.

Because doubly-linked lists are often used to keep track of similar data (for example, a process list), *Blacksheep* treats all elements of the list in aggregate. That is, *Blacksheep* keeps track of a single, representative, element for the entire list, assigning a category and subcategory to each dword that best describes dwords at that location in *every* element of the list (that is, it picks the most general type). For example, if a list has two elements, *A* and *B*, and the first dword of *A* is classified as *VALUE* whereas the first dword of *B* is classified as *ZERO*, *Blacksheep* will classify the first dword of the representative element as *VALUE*. *Blacksheep* uses the size of the dynamically allocated pools that the list elements are located in to determine the size of the list elements.

Naming. After generating a list of reachable memory locations (dwords) and their categories, *Blacksheep* assigns canonical names to each location to facilitate comparisons against other memory dumps. The names are assigned as follows:

For statically allocated data, *Blacksheep* generates a name consisting of the name of the module and the relative offset of the data element within that module. For example, a statically-allocated dword within a *ntoskrnl.exe*'s data segment (say, 102,088 bytes from the start of the driver) would be named "(ntoskrnl.exe+102,088)".

For doubly-linked list headers, *Blacksheep* generates a name consisting of the offset of the list header within its page in memory, the pool tag of the list header's pool (if available), the size of each element in the list, and the offset of the forward-link field within each list element. For example, a list header at offset 2,034 bytes into a pool tagged "NTKL," with elements of size 24, which have the forward link field at offset 8, would be named "(NTKL+2,034, 24, 8)".

For data in a linked list element, *Blacksheep* generates a name consisting of the offset of the list header name, and the offset within the element. Note that this is done only for the representative list element. For example, the dword at offset of 4 bytes into an element of the list "(NTKL+2,034, 24, 8)" would receive a canonical name of "(NTKL+2,034, 24, 8)+4".

Blacksheep considers changes in the category or subcategory of identically-named dwords between two memory dumps to be differences. For example, if "(ntoskrnl.exe+102,088)" is in category *ZERO* in one dump and *POINTER* in another, *Blacksheep* will count this dword as differing. We chose this granularity of comparison because more specific comparisons (for example, comparing the actual values of integers) resulted in unmanageable amounts of noise (benign differences) in the analysis. Likewise, any more general analysis quickly becomes meaningless.

The distance metric for data analysis is determined by the total number of dwords whose classification differs between the two dumps being compared. The differences that rootkits make to the data structures in memory cause such malicious dumps to stand out in this analysis.

4.4 Kernel Entry Points

Several mechanisms are used by Windows to switch execution from user-mode to kernel-mode and to handle hardware interrupts.

When an event triggers such a transition, a handler function pointer is loaded, and the kernel-mode execution begins at the location pointed to by that function pointer. We call these function pointers *kernel entry points*, since they are addresses where user-mode code “enters” the kernel.

The analysis of these pointers is useful for in rootkit detection tools, since rootkits frequently modify their values to allow rootkit functionality to be executed when a specific event occurs. This technique allows rootkits to subvert kernel behavior, in essence filtering kernel function invocations. *Blacksheep* checks the following kernel entry points.

Interrupt Description Table (IDT). The IDT is a hardware mechanism offered by an x86-compatible processor to allow the operating system to respond to interrupts. Windows only uses a limited set of interrupts, mapped the remaining interrupts to generic functions (named *nt!KiUnexpectedInterruptXX*, where XX is a number corresponding to the interrupt). In a non-infected system, all interrupts are mapped to kernel functions inside *ntoskrnl.exe* or *hal.dll* modules.

Interrupt *0x2E* is used to switch to kernel-mode when a system call is performed. Even though Windows uses the *SYSENTER* instruction as opposed to the IDT to switch to kernel mode on modern processors, this IDT entry is still set to *nt!KiSystemService*.

SYSENTER. The *SYSENTER* assembler instruction is used to quickly switch from user-mode to kernel-mode execution on modern x86 machines. When this instruction is called, the execution moves to an address that is stored in particular machine-specific registers (MSR). Windows sets these registers in such a way that whenever the *SYSENTER* instruction is executed, the kernel function named *nt!KiFastCallEntry* is called. This function, in turn, calls the requested system call according to the value stored in the *EAX* register and the currently active thread.

System Service Dispatch Table (SSDT). The SSDT is an array of virtual addresses, where each address is the entry point of a kernel function. When a kernel function is invoked, the function *nt!KiSystemService* reads this table and jumps to the required entry.

The address where the SSDT is located is specified on a per-thread basis in the *KTHREAD* data structure. Moreover, a thread can use more than one SSDT.

Usually, all threads share the same two SSDTs (*KiSystemService* for native Windows APIs, implemented by *ntoskrl.exe*, and *W32pServiceTable* for user and Graphical Display Interface functions, implemented by *win32k.sys*). However, a rootkit can create a new SSDT and modify a *KTHREAD* structure to make the associated thread use the new SSDT. Using this method, a rootkit can avoid being detected by tools checking only the two canonical SSDTs.

Call Gates. Call Gates are yet another mechanism to transfer control between x86 privilege levels. Call Gate descriptors are specified in the Global Descriptor Table (GDT), a data structure used by x86 processors that defines the characteristics of various memory areas.

Even though Call Gates are not normally used by modern operating systems, they can still be utilized by rootkits as a backdoor to enable the calling of kernel-mode functions from user-mode programs without the need of a persistent rootkit kernel module.

I/O request packet handlers. I/O request packets (IRPs) are kernel data structures used by Windows kernel modules to communicate with each other and with user-mode code. When a kernel module is loaded, an array of function pointers (one for each IRP type the module can handle) is initialized. Each of these functions is invoked when the corresponding I/O request is received by the kernel module.

Kernel entry point differences. Kernel entry points are computed by comparing the target addresses between the dumps. *Blacksheep* checks that each entry point points to the same offset within the same driver in both dumps. If the entry point points into dynamically-allocated memory, *Blacksheep* adds these memory sections to its code analysis. The distance metric that is calculated for the entry point analysis is the total number of such differences found between two dumps.

4.5 Clustering

In the clustering step, *Blacksheep* calculates a distance between every pair of dumps, creates hierarchical clusters based on this distance, and uses these clusters to classify the dumps.

4.5.1 Combined Distance

Blacksheep uses the four analyses previously described to calculate four differences between each memory dump pair. Our four analyses measure different things, which results in very different ranges of distance values. To combine our analyses results, *Blacksheep* first scales the distance values to a unit range (between 0 and 1). To this end, we find, for each of the four distances, the maximum distance value between any pairs of dumps. This maximum is used as the respective normalization factor. Once normalized, the four distance values are simply summed up, for a final distance value between each pair of memory dumps. While simple, this approach allows each analysis to contribute equally to the final distance, and our experiments show that it works well.

4.5.2 Clusters

Utilizing the distance metric, *Blacksheep* divides the memory dumps into clusters, using a standard, hierarchical clustering approach. We use the implementation provided by SciPy, with a "distance" linkage function. The threshold for the clustering step were derived manually, based on small scale experiments (we found that, overall, the distances between clean and infected dumps are typically noticeably larger than between two clean dumps).

Any generated clusters that contain less than a set threshold of memory dumps are marked as infected. This threshold is selected based on the size of the analyzed set, under the assumption that no more than a certain fraction of the dumps would be infected simultaneously. With modern attack patters, we feel that this assumption is a valid one. Specifically, a characteristic of APTs (Advanced Persistent Threats) is the compromise of a small amount of machines by an attacker in a stealthy manner. For example, in the Stuxnet attack on Irani nuclear reactors, malware was distributed over USB drives to a small amount of machines. Likewise, many examples of APTs starting with a spear-phishing campaign to infect a single machine have been documented. While *Blacksheep* would not be effective against a network worm that propels a rootkit throughout a crowd of machines, it would be effective against a compromise seeking to establish a foothold inside an organization.

Examining the clusters that *Blacksheep* produces can be informative for further analyses, as certain rootkit families reliably cluster together. This can provide valuable insight into tracking infections throughout an enterprise.

5. IMPLEMENTATION

We initially implemented *Blacksheep* for the analysis of memory acquired through either QEMU [3] introspection or through the use of a memory dumping driver. This allows *Blacksheep* to be used in both cloud computing and physical deployment scenarios.

The implementation of the *Blacksheep* approach consists of several phases. First, memory is acquired (by one of several methods as described in Section 5.1) and transferred over the network to our analysis server. Then, the analysis server submits jobs to distributed comparison workers, which generate comparisons between each pair of memory dumps. Finally, the clustering engine processes the comparison reports and generates clusters, and detect infections.

5.1 Memory acquisition

A variety of methods exist to acquire a dump of the physical memory (and, if applicable, a copy of the swap file) from a running machine. The method of choice affects the integrity and completeness of the dump and the possibility of evasion by rootkits. We will present a brief summary of these methods, and discuss their advantages and disadvantages. While the acquisition method affects the results, *Blacksheep* supports dumps acquired with any of these methods.

Software memory acquisition. Several tools exist with the purpose of acquiring physical memory dumps from Windows XP and Windows 7 operating systems. Such tools usually rely on accessing the physical memory via the *\\Device\PhysicalMemory* device present on these versions of Windows. Since the contents of physical memory are highly sensitive from a security perspective, modern versions of Windows restrict access to this device to kernel drivers only. This necessitates the creation and loading of kernel drivers to accomplish this task.

Additionally, acquiring the swap file of the system is another tricky task. Two software-based methods exist: (i) finding and cloning the handle to the swap file that Windows creates on startup (so that the handle can be later passed to userspace and the swap file read using standard Windows APIs), and (ii) parsing filesystem structures on the disk to copy the raw data directly. We chose the former approach, since it is independent of the underlying filesystem settings.

Memory dumps acquired in this way tend to contain a large amount of inconsistencies, for several reasons. First, the dumping driver and application itself must be loaded into memory, thereby modifying it. More importantly, however, is the fact that dumping memory by software is not an atomic operation, so the memory itself continues to be modified while the dumping procedure is performed. Additionally, such software can be easily tampered with by rootkits, as it runs with the same privilege levels and in the same memory space as the rootkit itself.

Despite the disadvantages, since all that is required is a software installation, this method is the easiest to deploy on a large scale.

Crash dumps and hibernation files. When a Windows system crashes or it is hibernated, the operating system saves a snapshot of the physical memory to disk. Once this occurs, the memory and swap can easily be read from the disk and utilized by *Blacksheep*.

Even if these methods are effective in creating memory dumps, they are not feasible for a widespread usage due to the fact that they require the system to be interrupted.

Physical devices. Hardware solutions have been proposed for dumping physical memory, exploiting the fact that external peripherals can utilize DMA to achieve direct access to system memory. In particular, hardware devices working on Firewire, PCI, PCIe, and ExpressCard interfaces are available.

This method does not need any running software on the target system, but some inconsistencies in the dumped memory are still possible if the system is not suspended while dumping the memory. Additionally, some specific memory locations, and the swap file, cannot be accessed by this method.

Techniques to avoid the dumping of some memory regions by hardware devices have been studied [28]. Such techniques depend on the specific hardware methods being used, and do not appear to have been utilized by rootkits as of yet. However, a software method might be helpful along with a hardware approach, to make sure that the dumps generated by the hardware device have not been tampered with.

This method is difficult to deploy on a wide scale due to the hardware requirement.

Virtual machine introspection. When a system is running inside a virtual machine, the virtualization software running on the host operating system can easily image the memory of the guest system. For instance, in QEMU, this is achieved through the use of the *pmemsave* command. While the dump is being captured, the virtual disk can be parsed to recover the swap file.

Dump artifacts are minimized because the dump is taken while the guest operating system is suspended. Minor inconsistencies are, however, still possible due to in-progress memory writes, especially in multi-processor systems.

Using virtual machine introspection, the dumping process cannot be tampered with by rootkits running on guest operating system, since the process runs on the host. However, a rootkit could use virtual machine detection techniques to modify or terminate its behavior when running inside a virtual machine, evading detection.

5.2 Dump Comparison

We utilize Volatility as a library to process Windows memory dumps in the comparison step. This allows us to support several different versions of Windows (specifically, Volatility supports 32-bit versions of Windows from Windows XP SP2 through Windows 7 SP1, and 64-bit support is planned as well), and abstracts away minute changes between Windows service packs and major releases. We have implemented parts of *Blacksheep* as Volatility plugins to be able to process swap memory, because this functionality was not available in Volatility.

Additionally, the *SYSENTER* target address described in Section 4.4 and utilized in the kernel entry point analysis is actually stored in the MSR 0x176 x86 CPU register. This is problematic due to the fact that for certain memory acquisition methods, such as the dumping of memory over DMA through the use of a hardware device, hardware registers (including MSR 0x176) are not saved. To surmount this obstacle, we have identified a location in memory where the Windows kernel stores an updated value of this register. For example, Windows XP SP3 keeps this value in the *KiTrap01* kernel function, as seen in Figure 1. However, this is system specific, and care has to be taken with regards to system upgrades when using such an acquisition method.

```
mov ecx, [ebp+68h]
cmp ecx, <copy of MSR 0x176 register>
```

Figure 1: The value of the MSR 0x176 register is stored in the *KiTrap01* kernel function in Windows XP SP3.

6. EVALUATION

We evaluated *Blacksheep* on two sets of memory dumps. The first was acquired from a set of Windows 7 virtual machines using QEMU VM introspection. Our virtual machines were images of Windows 7 on the same QEMU host system (so, we expect the operating system code to be identical in terms of hardware drivers and kernel modules). As we discussed in Section 5, the acquisition of memory images in this fashion from a virtual machine produces a very small amount of memory artifacts, and thus, this is the ideal setting for *Blacksheep*.

non-infected	non-infected	r2d2	tdss
non-infected	non-infected	r2d2	tdss
non-infected	non-infected	r2d2	tdss
non-infected	non-infected	r2d2	tdss
non-infected	non-infected	stuxnet	tld3
non-infected	non-infected	stuxnet	tld3
non-infected	non-infected	stuxnet	tld3
non-infected	non-infected	stuxnet	tld3
non-infected	non-infected	zeroaccess	zeroaccess
non-infected	non-infected	zeroaccess	zeroaccess

Table 1: Cluster results for the Windows 7 dataset with a clustering threshold of 1.8.

The second set of memory dumps was acquired using our memory acquisition driver from Windows XP machines (running on VirtualBox) to test the performance of *Blacksheep* on non-perfect dumps. The driver was used to acquire the memory dumps, and these dumps were transferred over the network to a central server. This method of acquisition produces many inconsistencies, and thus, introduces noise into *Blacksheep*'s analysis.

In addition, we have performed different, common tasks on the different machines (such as web browsing, working on Office documents, watching media files, ...) to ensure that the memory dumps are diverse.

We tested these configurations against a range of publicly available rootkits. In particular, we used the well-known Mebroot, Stuxnet, Rustock, and Blackenergy rootkits, two rootkits in the TDSS family (tdss and tdl3), and the r2d2 Trojan developed by the German government. Unfortunately, several existing rootkits do not function properly on Windows 7, so the range of tested rootkits is smaller for the first data set (the Windows 7 - QEMU data).

6.1 Windows 7 - QEMU Introspection

We tested *Blacksheep* against a set of 40 memory dumps taken through QEMU VM introspection. Within the set, 20 of the dumps were clean, and 20 were infected with rootkits, with 4 machines infected with each of 5 rootkits. After analyzing these dumps, *Blacksheep* generated a hierarchical clustering, shown as dendrogram in Figure 2. Based on the selected cluster distance threshold, two possible sets of clusters are shown in Table 1.

The detection rate of *Blacksheep* depends on the threshold chosen in the clustering step. After producing clusters, all clusters of size 4 or less were tagged as malicious. This is because we expect that benign dumps group together, while infected dumps form outliers, and that infected dumps will not account for more than 10% of the installed machines in an organization. With a threshold of 1.8, *Blacksheep* achieves a true positive rate of 100%, and a false positive rate of 0%. As expected, all rootkits cluster together with other rootkits in their families. This is because of the similar differences that these rootkits introduce into the kernel code and data.

6.2 Windows XP - Driver-acquired Memory

Blacksheep was also tested in detecting rootkits on Windows XP. Again, 10 clean dumps were clustered, this time together with 8 rootkits. The hierarchical clustering results are shown in Figure 3, and resulting clusters are shown in Table 2. Again, all clusters of size one were tagged as malicious.

With a clustering threshold of 0.6, *Blacksheep* produced 62.5% true positives and 0% false positives, and with a clustering threshold of 0.4, *Blacksheep* produced 75% true positives and 5.5% false positives.

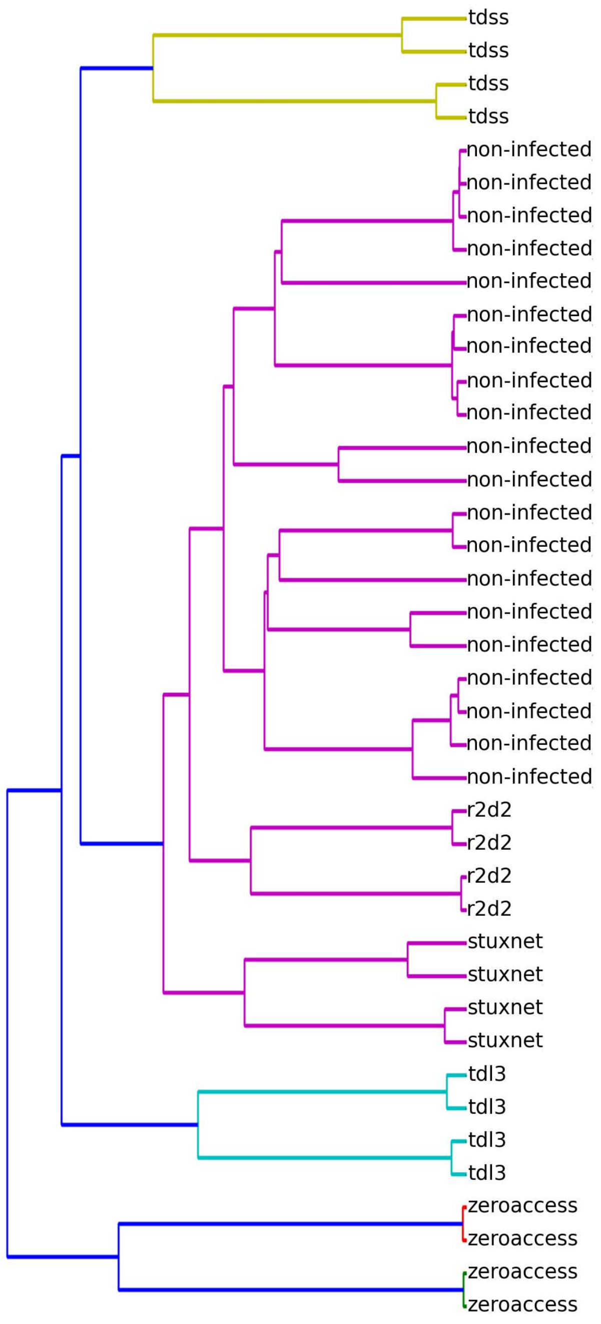

Figure 2: Hierarchical cluster dendrogram for the Windows 7 dataset.

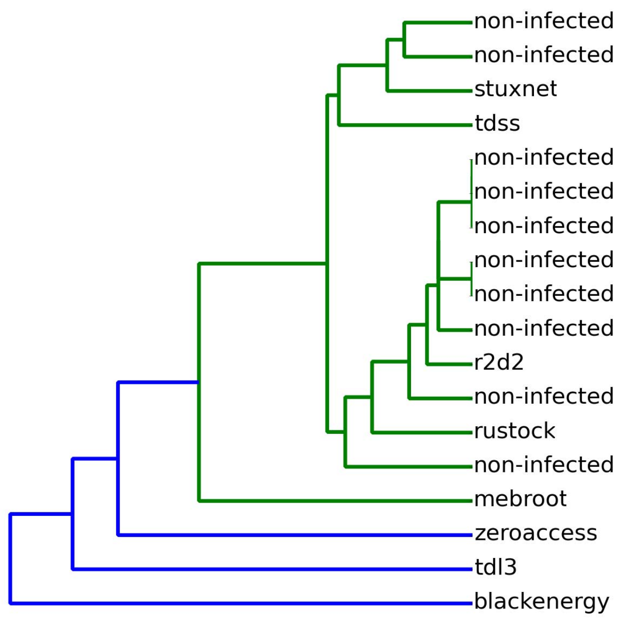

Figure 3: Hierarchical cluster dendrogram for the Windows XP dataset.

rustock	rustock
r2d2	r2d2
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
non-infected	non-infected
stuxnet	stuxnet
non-infected	non-infected
non-infected	non-infected
tdss	tdss
mebroot	mebroot
zeroaccess	zeroaccess
tdl3	tdl3
blackenergy	blackenergy

Table 2: Cluster results for the Windows XP dataset with a clustering threshold of 0.5 (left) and 0.4 (right).

These results demonstrate how critical consistent memory dumps are to *Blacksheep*'s operation. Specifically, while configuration analysis, code analysis, and entry point analysis performed as well as for the QEMU images, data comparison suffered. This is due to kernel data continuing to change while the dump is being acquired, creating inconsistencies in the final image. In contrast, the code and entry point sections of kernel memory are considerably more stable.

6.3 Performance

The runtime performance of *Blacksheep* depends on several factors, including the size of the memory dumps, the size (or absence) of the swap file, and the hardware involved. For memory dumps of one gigabyte of RAM, we were able to compute the differences between a pair of memory dumps in 10 minutes. The hierarchical clustering requires $O(n^2)$ comparisons, but the results are cached so that after the initial clustering, every new dump will require $O(n)$ comparisons to recompute the clusters. The comparisons themselves are trivial to parallelize, and the clustering step is computed very quickly, so *Blacksheep* can be horizontally scaled to linearly increase performance.

7. DISCUSSION

In this section, we discuss the limitations of our approach. *Blacksheep* relies on two main assumptions: i) it is possible to collect comparable memory dumps, and ii) rootkit infections modify memory dumps in a detectable way. When either of these assumptions is violated, the approach implemented by *Blacksheep* fails.

The first issue is related to how and when the dumps are collected. To maximize the homogeneity among memory dumps, it would be best to collect memory snapshot in similar states (and at similar times) across all hosts. Unfortunately, it is not always easy to determine "checkpoints" that are comparable across machines, and, therefore, it can often be the case that memory dumps are collected at very different times, and in very different states, resulting in unwanted differences in memory layout and contents.

Furthermore, it is extremely important to minimize the number of inconsistencies in the analyzed memory dumps. This is an especially challenging problem when acquiring memory from physical hardware, as the various acquisition methods detailed in 5 all have various drawbacks.

Virtualization and cloud-based systems offer an ideal setting for the collection of memory dumps, as many virtualization environments offer the ability to take snapshots of the guest operating system at well-defined times, which improves the chances of collecting homogeneous images. In addition, virtualized hardware can offer a level of homogeneity that real hardware would not be able to achieve, as real hardware can fail and might be substituted with different lines of products, which, in turn, might require different drivers.

The second issue is related to the way in which rootkits affect the layout and contents of memory. Rootkits could attempt to evade detection by modifying parts of the kernel memory that, by design, change frequently across machines. These high-entropy areas cannot be used as a basis to determine the crowd invariants, and, therefore, represent an opportunity for evasion. Our technique cannot detect this type of rootkits. However, the implementation of such system would be very challenging (and, in fact, there are no known instances of such malware), because the same unpredictability that makes deriving invariant difficult, would likely make a rootkit unstable.

Another problem is the process of updating a crowd of similar computers, which might introduce changes that are mis-detected as an infection. In this case, *Blacksheep* would need to be disabled until a sizable amount of machines are updated, and a sufficient baseline re-established.

Finally, it is important to note that *Blacksheep*'s approach is agnostic to memory location randomization techniques such as ASLR. This is due to the fact that *Blacksheep* compensates for relocation in its code analysis, and uses relative memory locations for its data and entry point analyses.

8. CONCLUSIONS

In this paper, we have described *Blacksheep*, a novel system designed to detect kernel-level rootkit infestations in a crowd of similarly-configured machines. We have discussed the current state of the art in the field, argued why *Blacksheep* extends it, and presented the results of our analyses. We feel that *Blacksheep* would be an useful tool for organizations with the right population of machines, as such organizations can greatly benefit from *Blacksheep*'s ability to recognize existing infections and 0-days and its ease of administration compared to present security offerings.

Furthermore, we have offered the insight into some internal workings of the Windows kernel in the hopes that it would be useful to the scientific community.

Acknowledgments: This work was supported by the Office of Naval Research (ONR) under Grant N000140911042 and the National Science Foundation (NSF) under grants CNS-0845559 and CNS-0905537.

9. REFERENCES

[1] Gmer. http://www.gmer.net/, May 2012.

[2] Hbgary responder pro. http://www.hbgary.com/responder-pro-2, May 2012.

[3] Qemu website. http://qemu.org, May 2012.

[4] Windows academic program. http://www.microsoft.com/education/facultyconnection/articles/articledetails.aspx?cid=2416, Apr. 2012.

[5] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, Vol. 8, No. 5, Sept. 2010.

[6] B. Blunden. *The Rootkit Arsenal*. Wordware Publishing, 2009. Chapter 7.9.

[7] M. Burdach. Finding digital evidence in physical memory. In *Black Hat Federal Conference*, 2006.

[8] M. Carbone, W. Lee, W. Cui, M. Peinado, L. Lu, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *ACM Conf. on Computer and Communications Security*, 2009.

[9] B. Cogswell and M. Russinovich. Rootkitrevealer. http://technet.microsoft.com/en-us/sysinternals/bb897445, Nov. 2008.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69, Dec. 2007.

[11] F. Gadaleta, N. Nikiforakis, J. Mühlberg, and W. Joosen. Hyperforce: Hypervisor-enforced execution of security-critical code. *Information Security and Privacy Research*, pages 126–137, 2012.

[12] F. Gadaleta, N. Nikiforakis, Y. Younan, and W. Joosen. Hello rootkitty: a lightweight invariance-enforcing framework. *Information Security*, pages 213–228, 2011.

[13] G. L. Garcia. Forensic physical memory analysis: an overview of tools and techniques. In *TKK T- 110.5290 Seminar on Network Security*, 2007.

[14] K. Griffin, S. Schneider, X. Hu, and T. cker Chiueh. Automatic generation of string signatures for malware detection.

[15] G. Hoglund. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[16] G. Jacob, H. Debar, and E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4:251–266, 2008. 10.1007/s11416-008-0086-0.

[17] A. Kapoor and R. Mathur. Predicting the future of stealth attacks. *Virus Bulletin conference*, Oct. 2011.

[18] J. D. Kornblum. Exploiting the rootkit paradox with windows memory analysis. *International Journal of Digital Evidence*, 2006.

[19] J. D. Kornblum. Using every part of the buffalo in windows memory analysis. *Digital Investigation*, Mar. 2007.

[20] Z. Li, M. Sanghi, Y. Chen, M. yang Kao, and B. Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 32–47. IEEE Computer Society, 2006.

[21] M. H. Ligh. Volatility malware plugins. http://code.google.com/p/malwarecookbook.

[22] Z. Lin, J. Rhee, X. Zhang, D. Xu, and X. Jiang. Siggraph: Brute force scanning of kernel data structure instances using graph-based signatures. In *the 17th Network and Distributed System Security Symposium*, 2011.

[23] McAfee. Mcafee deepsafe. http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx, 2011.

[24] Microsoft. Kernel patch protection: Faq. http://msdn.microsoft.com/en-us/windows/hardware/gg487353, Sept. 2007.

[25] N. L. Petroni, J. Timothy, F. Aaron, W. William, and A. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proceedings of the USENIX Security Symposium*, pages 289–304, 2006.

[26] M. E. Russinovich and D. A. Solomon. *Windows Internals*. Microsoft, 5th edition, June 2009.

[27] J. Rutkowska. Rootkits vs. stealth by design malware. https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Rutkowska.pdf, 2006.

[28] J. Rutkowska. Beyond the cpu: Defeating hardware based ram acquisition (part i: Amd case). In *Black Hat DC*, 2007.

[29] A. Schuster. Pool allocations as an information source in windows memory forensics. In *Pool Allocations as an Information Source in Windows Memory Forensics*, 2006.

[30] A. Schuster. Searching for processes and threads in microsoft windows memory dumps. In *Digital Investigation*, 2006.

[31] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses, 2007.

[32] R. Treit. Some observations on rootkits. http://blogs.technet.com/b/mmpc/archive/2010/01/07/some-observations-on-rootkits.aspx, Jan. 2010.

[33] D. Wagner. Mimicry attacks on host-based intrusion detection systems. *Proceedings of the 9th ACM conference on computer and communications security*, 2002.

[34] A. Walters. The volatility framework: Volatile memory artifact extraction utility framework. https://www. volatilesystems.com/default/volatility.

[35] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *ACM Conf. on Computer and Communications Security*, Nov. 2009.

[36] Y. Xie, H. Kim, D. O'Hallaron, M. Reiter, and H. Zhang. Seurat: A pointillist approach to anomaly detection. In *Recent Advances in Intrusion Detection*, pages 238–257. Springer, 2004.

[37] H. Yin, P. Poosankam, S. Hanna, and D. Song. Hookscout: Proactive binary-centric hook detection. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Bonn, Germany*, July 2010.