Hypervisor-based Malware Protection with AccessMiner

Aristide Fattori†, Andrea Lanzi†, Davide Balzarotti‡, Engin Kirda¶
{aristide.fattori,andrea.lanzi}@unimi.it, balzarotti@eurecom.fr, ek@ccs.neu.edu
† Università degli Studi di Milano, ‡Eurecom, ¶Northeastern University

Abstract—In this paper we discuss the design and implementation of AccessMiner, a system-centric behavioral malware detector. Our system is designed to model the general interactions between benign programs and the underlying operating system (OS). In this way, AccessMiner is able to capture which, and how, OS resources are used by normal applications and detect anomalous behavior in real-time.

The advantage of our approach is that it does not require to be trained on malicious samples, and therefore it is able to provide a general detection solution that can be used to protect against both known and unknown malware. To make the system more resilient against tampering from sophisticated attackers, AccessMiner is implemented as a custom hypervisor that sits below the operating system. In this paper we discuss the implementation details and the technical solutions we adopted to optimize the performances and reduce the impact of the system.

Our experiments show that in a stable environment AccessMiner can provide a high level of protection (around 90% detection rate with zero false positives) with an acceptable overhead - similar to the one that can be experienced in a state of the art virtual machine environment.

Keywords: Malware detection, OS protection, Behavioral-based detection, Hypervisor

Curresponding Author: Aristide Fattori
Address: Dipartimento di Informatica, Via Comelico 39, 20135, Milano, Italy
Telephone: +39 0250316362

1 Introduction

The problem of detecting attacks and malicious applications at the host level has been largely studied by both the research and the industrial communities. The most common solutions are based either on matching static signatures or on using behavioral models to specify allowed or forbidden behaviors. Signatures work well to identify single malware instance but they quickly become ineffective when the attacker adopts obfuscated or polymorphic code. At the same time, most behavior-based detection techniques follow a *program-centric* approach that focuses on modeling the execution of individual programs. These models often lack the context to capture how *generic* benign and malicious programs interact with their environment and with the underlying operating system. As a result, detectors based on program-centric behavioral techniques tend to raise alerts whenever a new program is encountered or an existing program is used in a different way. This typically leads to unacceptably high false positive rates—thus limiting the practical applicability of these approaches.

AccessMiner [Lanzi et al., 2010] introduced a novel *system-centric* technique to model the activity of benign programs. The main idea behind the AccessMiner approach is that, given enough training, it is possible to identify common patterns in the way benign applications interact with the operating system resources. For instance, while normal programs typically write only to their own directories (and to temporary directories), malware often attempt to tamper with other applications and critical system settings, often residing outside the normal application "scope". As a result, special *access activity models* can be derived by AccessMiner only by looking at the execution of a broad set of benign applications. Therefore, traces of malware execution, often problematic to collect from a coverage and diversity point of view, are not required to train our classifiers.

While our experiments showed that a system-centric approach was successful in identifying a large amount of diverse malware samples with very few false positives, a number of important points were not addressed in the original paper. In particular, the original approach was designed to be implemented as part of the Windows operating system kernel. However, the threat scenario rapidly changed in the last years with the creation of new attacks techniques (i.e. Rootkits) which

aim is to protect user-space malware from detection models and disable security mechanisms (i.e. Reference Monitors). Rootkits has always been shipped with two main components: a kernel-level component and user-space component. The goal of the former is to disable security mechanism and hide information from the system, while the aim of the second is to perform malicious actions. Both components are essential for successful, simple and general attack design. Moreover the rise of targeted attacks also poses new challenges that are not fully addressed by current methodologies. For example, by carefully combining a mix of social engineering, zero days exploits for unknown Windows vulnerabilities, and stolen certificates to sign kernel modules - motivated and well-funded attackers can quickly subvert the target OS and remain undetected for long period of times (as the Stuxnet [Symantec, 2011b], Duqu [Symantec, 2011a], and Flame [Symantec, 2012] incidents have shown).

Since a successful targeted attack could easily tamper with OS-based detection mechanisms, in this paper we re-design AccessMiner and we describe how the same approach can be implemented as a custom hypervisor. This new solution makes the detector much more resilient to sophisticated attacks (Rootkits that tries to disable the Reference Monitors), but it also introduces several technical problems and challenges. First, in order to collect the same information and monitor the system calls issued by each process, a hypervisor has to solve the so-called *semantic gap* and it has to provide a trusted path related to the system call invocation. Even though many solutions exists for this problem, current hypervisor-based detection countermeasures do not scale well to several scenarios (e.g., critical infrastructures), due to their high computational requirements that conflict with the strict timing constraints of the running applications. The challenge here is to use a lightweight approach that does not impact the performance of the system in a prohibitive way.

To summarize, the new contributions of this paper are the following:

- We extend and complement the original AccessMiner paper by presenting a real implementation as a custom hypervisor. In particular we design a system that protect itself from sophisticated attacks techniques (i.e. Rootkits) and provide at the same time a trusted path for the system call invocation as a main source of information for our detection system. We describe the problems we had to face and the solutions we developed to adapt the original algorithm to this setup.
- We extensively tested the new detector, in particular to show to which extent it affects the performance of the system. Our experiments show that the high protection provided by AccessMiner can be obtained at the price that is normally paid by running a system inside a state-of-the-art virtual environment, such as the ones normally adopted in the cloud.

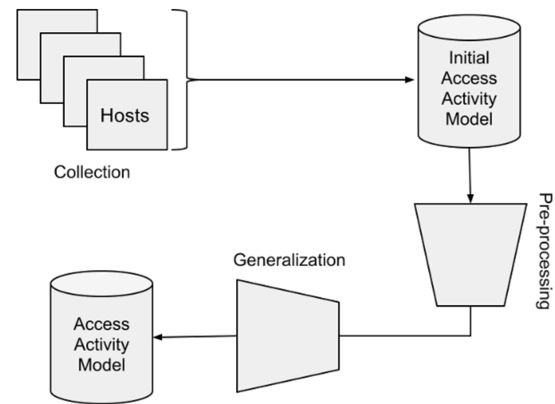

Fig. 1. Creation of System-centric models.

2 System-Centric Models and Detection

Several studies [Lanzi et al., 2010], [Canali et al., 2012] have shown that models based on system call sequences (n-grams) have difficulties in distinguishing normal and malicious behaviors. One of the main problems is that while n-grams might capture well the execution of individual programs, they poorly generalize to other applications. The reason is that the model is closely tied to the execution(s) of particular applications; we refer to this as a program-centric detection approach.

In this section, we propose a model that attempts to abstract from individual program runs and that generalizes how benign programs generally interact with the operating system. For capturing these interactions, we focus on the file system and the registry activity of Microsoft Windows processes. More precisely, we record the files and the registry entries that Windows processes read, write, and execute (in case of files only).

Our model is based on a large number of runs of a diverse set of applications, and it combines the observations into a single model that reflects the activities of *all* programs that are observed. For this to work, we leverage the fact that we see “convergence.” That is, even when we build a model from a subset of the observed processes, the activity of the remaining processes fits this model very well. Thus, by looking at program activity from a system-centric view – that is, by analyzing how benign programs interact with the OS – we can build a model that captures well the activity of these programs. Of course, this would not be sufficient by itself. To be useful, our model must also be able to identify a reasonably large fraction of malware. To demonstrate that this is indeed the case, we have performed a number of experiments that are described in more detail in Section 5.

Figure 1 captures the creation of the access activity model. All the steps that are required for the creation are explained in the following of this section.

2.1 Creating Access Activity Models

To capture normal (benign) interactions with the file system and the Windows registry, we propose the creation of *access*

activity models. An access activity model specifies a set of labels for operating system resources. In our case, the OS resources are directories in the file system and sub-keys in the registry (sub-keys are the equivalent of directories in the file system). For simplicity, in the following we refer to directories and sub-keys as "folders".

Note that we do not specify labels directly on files or registry entries. The reason for this was that the resulting models are significantly smaller when looking at folders only. As a result, the model generation process is faster and "converges" quicker (i.e., less training data is required to build stable models). Moreover, in almost all cases, the labels for the folder entries (files or registry keys) would be similar to the label for that folder itself. Thus, the sacrifice in precision is minimal.

A label L is a set of access tokens $\{t_0, t_1, \ldots, t_n\}$. Each token t is a pair $\langle a, op \rangle$. The first component a represents the application that has performed the access, the second component *op* represents the operation itself (that is, the type of access).

In our current system, we refer to applications by name. In principle, this could be exploited by a malware process that decides to reuse the name of an existing application (that has certain privileges). In the future, we could replace application names by names that include the full path, the hash of the code that is being executed, or any other mechanism that allows us to determine the identity of the application that a process belongs to. Such techniques are already described in different papers [Litty et al., 2008], and its implementation is out of the scope of this paper. In addition to specific application names, we use the star character ($*$) as a wildcard to match any application.

The possible values for the operation component of an access token are read, write, and execute for file-system resources (directories), and read and write for registry sub-keys.

2.2 Initial Access Activity Model

An initial access activity model precisely reflects all resource accesses that appear in the system-call traces of all benign processes that we monitored (we call this data set the training data). Note that for this, we merge accesses to resources that are found in different traces and even on different Windows installations. In other words, we build a "virtual" file system and registry that contains the union of the resources accessed in all traces.

Whenever an application proc opens or reads from an existing file foo in directory C:\path\dir, we insert the directory dir into our "virtual" file system, including all directories on the path to dir. When a prefix of the directories along path already exist in our virtual file system, then these directories are re-used. All directories that are not already present (including dir) are added to the virtual file system tree. Then, we add the access token $\langle \mathsf{proc}, \mathsf{read} \rangle$ to the label associated with dir.

When a process creates or deletes a file in a directory dir, or when it writes to a file, then we use the operation write for the access token. Similar considerations apply for read and write operations that are performed on the registry. Finally, whenever a binary is executed (loaded by the OS loader), then we add a token with execute to the directory that stores this binary.

For example, consider that file C:\dir\foo is read by pA on machine A, and that file C:\dir\sub\bar is written by pB on another machine B. Then, the resulting virtual file system tree would have C:\ as its root node. From there, we have a link to the directory dir, which in turn has a link to sub. The label associated with dir is $\{\langle \mathsf{pA}, \mathsf{read} \rangle\}$, and the label associated with sub is $\{\langle \mathsf{pB}, \mathsf{write} \rangle\}$.

2.3 Pre-Processing

Before the model generation can proceed, there are two additional pre-processing steps that are necessary. First, we need to remove a small set of benign processes that either read or execute files in many folders. The problem is that these applications appear in many labels and could lead to an access activity model that is less tight (restrictive) than desirable. We found that such applications fall into three categories: Microsoft Windows services (such as Windows Explorer or the command shell) that are used to browse the file system and launch applications; desktop indexing programs; and anti-virus software. The number of different applications that belong to these categories is likely small enough so that a manually-created white list could cover them. In our system, we remove all applications that read or execute files in more than ten percent of the directories. We found a total of 15 applications that fit this profile: nine Windows core services, two desktop indexing applications, and six anti-virus (AV) programs. Identifying such applications automatically is reasonable, because we assume that our training data does not contain malicious code. However, the number of white-listed applications is so small that the entries can be easily verified manually.

The second pre-processing step is needed to identify applications that start processes with different names. We consider that two processes with different names belong to the same application when their executables are located in the same directory. We have found 14 applications that start multiple processes with different names. These include well-known applications such as MS Office, Messenger, Skype, and RealPlayer. Of course, all Windows programs that are located in C:\Windows\system32 are also aggregated (into a single meta-application that we refer to as win_core). Merging processes that have different names but that ultimately belong to the same application is useful to create tighter access activity models.

2.4 Model Generalization

Based on the initial access activity model, we perform a generalization step. This is needed because we clearly cannot assume that the training data contains all possible programs that can be installed on a Windows system, nor do we want to assume that we see all possible resource accesses of the applications that we observed. Also, the initial model does

not contain labels for all folders (recall that the access is only recorded for the folder that contains the accessed entity).

The generalization step performs a post-order traversal of both the virtual file system tree and the virtual registry tree. Whenever the algorithm visits a node, it performs the following four steps:

Step 1: First, the algorithm checks the children of the current node to determine whether access tokens can be *propagated* upward in the tree. Intuitively, the idea is that whenever we inspect a folder (node) and observe that all its sub-folders are accessed by a single application only, we assume that the current folder also belongs to this application.

More formally, the upward propagation rule works as follows: For each operation op, we examine the labels of all child nodes and extract the access tokens that are related to op. This yields a set of access tokens $\{t_1, \ldots, t_n\}$. We then inspect the applications involved in these accesses (i.e., the first component of each token t_i). When we find that all accesses were performed by a *single* application proc, we add the access token $\langle \mathsf{proc}, \mathsf{op} \rangle$ to the current label.

Step 2: The upward propagation rule of Step 1 is used to identify parts of the file system or the registry that belong to a single application. However, this is problematic when considering *container* folders. A container is typically a directory that holds many "private" folders of different applications. A private folder is a folder that is accessed by a single application only (including all its sub-folders). A well-known example of a container is the directory C:\Program Files, which stores the directories of many Windows programs.

Since a container holds folders owned by many different applications, its label would deny access to all sub-folders that were not seen during training. This might be more restrictive than necessary. In particular, we would like to ensure that whenever an application accesses a previously-unseen folder in a container, this should be allowed. Intuitively, the reason is that this access follows an expected "pattern," but the specific folder has not been seen during training. To handle these cases, we introduce a special flag that can be set to mark a folder as a container.

The following rule is used to mark a folder as a container: Similar to before, we examine the labels of all child nodes and extract the access tokens that are related to each operation op. We then inspect the set of access tokens that is extracted $\{t_1, \ldots, t_n\}$. When the applications in these accesses are different, but there is *no wildcard* present in any access token, then the folder is marked as container. We explain the implications of a *container flag* for detection in Section 2.5.

Step 3: Next, the access tokens in the label associated with the current node are *merged*. To this end, the algorithm first finds all access tokens that share the same operation op (second component). Then, it checks their application names (first components). When all tokens share the same application name, they are all identical, and we keep a single copy. When the application names are different, or one token contains the wildcard, then the tokens are replaced by a single token in the form $\{\langle *, \mathsf{op} \rangle\}$. Merging is useful to generalize cases in which we have seen multiple applications that perform identical operations in a particular folder, and we assume that other applications (which we have not seen) are also permitted similar access.

Step 4: Finally, the algorithm adds access tokens that were likely missed because of the fact that the training data is not complete. More precisely, for each access token that is related to a *write* operation, we check whether there exists a corresponding *read* token. That is, for all applications that have written to a folder, we check whether they have also performed read operations. If no such token can be found, we add it to the label. The rationale for this step is that an application that can write to resources in a folder can very likely also perform read operations. While it is possible to configure files and directories for write-only access, this is very rare. On the other hand, adding read tokens allows us to avoid false positives in the more frequent case where we have simply not seen (legitimate) read operations in the training data.

When the generalization algorithm completes, all nodes in the virtual file system and the registry tree have a (possibly empty) label associated with them.

Note that, for building the access activity model, we do not require any knowledge about malicious processes. That is, the model is solely built from generalizing observed, good behavior.

2.5 Model Enforcement and Detection

Once an access activity model M is built, we can deploy it in a detector. More precisely, a detector can use M to check processes that attempt to read, write, or execute files in directories or that read or write keys from the registry.

The basic detection algorithm is simple. Assume that an application proc attempts to perform operation op on resource r located in \path\dir. We first find the longest prefix P shared between the path to the resource (i.e., \path\dir) and the folders in the virtual tree stored by M. For example, when the virtual file system tree contains the directory C:\dir\sub\foo and the accessed resource is located in C:\dir\sub\bar, the longest common prefix P would be C:\dir\sub. We then retrieve the label L_P associated with this prefix and check for all access tokens that are related to operation op (actually, after generalization, there will be at most one such token, or none). When no token is found, the model raises an alert. When a token is found, its first component is compared with proc. When the application names match or when the first component is $*$, the access succeeds. Otherwise, an alert is raised.

The situation is slightly more complicated when the folder that corresponds to the prefix P is marked as *container*. In this case, we have the situation that a process accesses a sub-folder of a container that was not present in the training data. For example, this could be a program installed under C:\Program Files that was not seen during training. In this case, the access is *permitted*. Moreover, the model is dynamically extended with the full path to the resource, and all new folders receive labels that indicate that application proc is its owner. More precisely, we add to each label access tokens in

form of $\langle \mathsf{proc}, \mathsf{op} \rangle$ for all operations. This ensures that from now on, no other process can access these newly "discovered" folders. This makes sense, because it reflects the semantics of a container (which is a folder that stores sub-folders that are only accessed by their respective owners).

Whenever an alert is raised, we have several options. It is possible to simply log the event, deny that particular access, or terminate the offending process.

3 Hypervisor Framework Design

In this section we present the design of a hypervisor-based detector that implements the system centric technique presented in the previous section. It is important to note that the design part of the Hypervisor is one of the main contributions of the extension of the paper.

Our enforcement model exploits hardware virtualization support available in commodity x86 CPU [AMD, Inc.,], [Neiger et al., 2006]. Leveraging hardware-assisted virtualization technology, we design a tamper-resistant and efficient detector that is able to take over the OS operations and verify the policies derived from the AccessMiner system. Our design goals provide two main contributions: (1) Provide an efficient detection monitor technique (2) Establish a trusted path for system call execution and provide control flow integrity for the whole system call execution. Both properties are really important to design a resilient and secure reference monitor.

3.1 Technology Overview

Before presenting the details of our detector implementation, we provide a brief introduction on Intel Virtualization Technology (VT-x) [Neiger et al., 2006].

The main characteristic of Intel VT-x is the support for two new VMX modes of operation. When VMX is enabled, the processor can be either in *VMX root mode* or in *VMX non-root* mode. The behavior of the processor in VMX root mode is similar to classic protected mode, except for the availability of a new set of instructions, called VMX instructions. Non-root mode is, instead, limited, even when the CPU is running in ring 0. Thanks to this, the virtual machine monitor (VMM) can inspect and intercept operations on critical resources without modifying the code of the guest OS (i.e., the *virtualized* OS). Moreover, because non-root mode operation supports all four IA-32 privilege levels, guest software can run in the original ring it was designed for.

A processor which has been turned on in normal mode can be switched to VMX root operation by executing a `vmxon` operation. The VMM running in root mode sets up the environment and initiates the virtual machine by executing the `vmlaunch` instruction.

Intel VT-x technology defines a data structure called virtual machine control structure (VMCS) that embeds all the information and the configuration needed to capture the state of the virtual machine, or resume its execution. The various control fields determine the conditions under which control leaves the virtual machine (VM exit) and returns to the VMM, and define the actions that need to be performed during VM entry and VM exit operations.

Various events may cause a VM exit, and can be configured with a very fine precision by the hypervisor (e.g., exceptions, I/O operations). Furthermore, the processor can also exit from the virtual machine *explicitly* by executing a `vmcall` instruction.

3.2 Threat Model

The threat model we adopt in this paper considers a very powerful attacker who can operate with kernel-level privileges. On the other side, the attacker does not have physical access to the machine and, therefore, cannot perform any hardware-based attack (e.g., a DMA attack [Wojtczuk, 2008]) and he cannot tamper with the hypervisor operations. We assume that our hypervisor starts during the boot process of the machine and it is the most privileged hypervisor on the system.

Should the deployment scenario require it, it is also possible to leverage *late-launching* [Neiger et al., 2006] to load AccessMiner hypervisor after the boot. For this to be feasible, however, we have to relax our threat model a little. Indeed, we must assume that either there is no malware on the machine *before* we launch AccessMiner or that we leverage an integrity checking technique to ensure that the hypervisor is not altered at load time [Martignoni et al., 2010], [McCune et al., 2008]. Despite this requirement, a hot-bootable hypervisor can be quite useful in scenarios in which it is not possible to restart the machine (e.g., when it provides some critical service).

3.3 Hypervisor Architecture

The Detector system is composed by three components: a *system call interceptor*, a *policy matcher*, and a *process revealer*. The outputs of all the components are combined together to check the policies derived by AccessMiner system. In Figure 2, we depict a scheme of the overall architecture. The overhead in executing security tools out of the guest OS is primarily due to the change in privilege levels that occurs while switching back and forth between the kernel-level and the hypervisor-level. We set the performance requirements for system call tracer's design to improve the performance of the system. In particular, we set two properties:

- (P1) **Fast invocation**: Invoking the monitor handler for a system call should not involve any privilege level changes if it is not needed.
- (P2) **Data read/write at native speed:** The monitor code should be able to read and write any system data and local data at native speed.

To state the security requirements, we consider an adversarial program A residing in the same environment as the system P. As we already described above In our threat model, A runs with the highest privilege in the guest VM and therefore can directly read from, write to and execute from any memory location that is not protected by the hypervisor by using sophisticated attack such as kernel rootkits. To ensure the security of the system call Monitor and its own trusted path, we state the security requirements:

- **(S1) Isolation of Monitor's code and data:** This ensures the integrity of the Monitor's code and data is protected

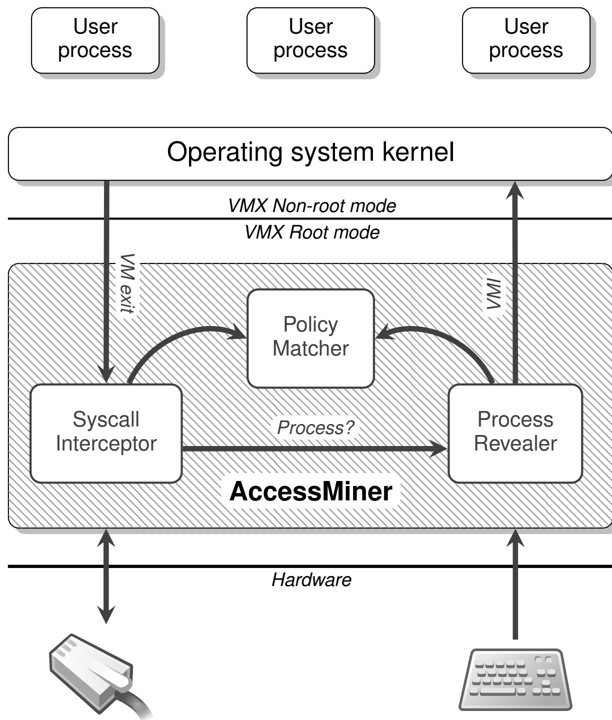

Fig. 2. Hypervisor Architecture.

from the adversary A. The Hypervisor approaches satisfy this requirement because A does not have any means to access to the Hypervisor code and data.

- **(S2) Designated point for switching into Monitor's code:** This requirement ensures that an attacker does not invoke any code in Monitor's code other than the designated points of entry (i.e., system call invocation).
- **(S3) A handler is called if and only if the corresponding hook is executed:** This requirement has two parts:
  - If a hook is reached in the monitored system, then the corresponding handler must be initiated by the system.
  - A handler is initiated only if the hook was executed.

  The second requirement can be satisfied because the exact vmcalls that initiated the hypervisor execution can be identified and checked.
- **(S4) The hooking mechanism must provide a trusted path system call execution:** The interception mechanism must provide a trusted path between the invocation of the system call and its own termination.

Our interception mechanism is designed with all the performance and security requirements in mind, described above.

3.3.1 Protected Memory Mechanism

Generally, the kernel is mapped into a fixed address range in each process address space. We define this address range the system address space. Since we are primarily interested in kernel level monitoring (e.g., system call parameters), we denote any code and data contained in the system address space as kernel code and kernel data. Since we do not know which kernel pages will contain data (e.g., some kernel code pages could be re-mapped in data pages) we need to map all the kernel memory pages into the Monitor address space. All pages containing kernel code will have read and execute privileges, but we assume that the kernel code can be write protected. The data regions will have all access rights. The System call Monitor (SCM) address space includes the Monitor's code (SCM code) and data (SCM data). However, some of the permissions are set differently. The kernel code and data regions do not have execute permissions. This means that while execution is within the SCM address space, no code mapped from the system address space will be executable. This is used to limit the surface attack code in case a privilege escalation attack occurs at the hypervisor level. The invocation checking modules are also contained only in the SCM address space and have execution privileges. Since the system address space contents are mapped into the SCM address space, an important requirement for the mapping to work is to ensure that other additional regions in the SCM address space do not overlap with the mapped regions from the system address space. This is achieved during the initialization of the Hypervisor that splits the memory in two main areas, the one dedicate to the system and the one dedicated to the Hypervisor data and code. It is important to note that the Hypervisor for the system should only map the entries of the page table that refers to the Data and Kernel Code. Since the System call Monitor address space contains all kernel data and also the Monitor data in its address space, the instructions as part of the security monitor can access these regions at native speed. This satisfies the performance requirement (P2). The memory mapping method we have introduced satisfies the isolation security requirement (S1) by having the Monitor code and data regions in a separate Monitor address space.

3.3.2 Checking Invocation Points

The entry and exit gates are the only regions that are mapped into both the system address space in pages having executable privilege. This ensures that a transfer between the address spaces (system to SCM and vice versa) can only happen through code contained in these pages. Moreover, since these pages are write-protected by the hypervisor, its contents cannot be modified by any in-guest code.

To satisfy the security requirement (S3, S4), once the SCM address space is entered through one of the entry gates, the invocation of the gate needs to be checked to ensure that it was from the only hook that is allowed to call the gate. The challenge is that, since the entry gate is visible to the guest OS's system address space, a branch instruction can jump to this location from anywhere within the system address space. Moreover, we cannot rely on call instructions and checking the call stack because they are within the system address space and as such the information cannot be trusted. We utilize a hardware debugging feature available in the Intel processors after Pentium 4 to check the invocation points. This feature, which is called last branch recording (LBR), stores the sources and targets of the most recently executed branch instructions in some specific processor registers. The last branch recording feature is activated by setting LBR flag in the `IA32_DEBUGCTL` MSR. Once set, the processor records

a running trace of a fixed number of last branches executed in a circular queue.

For each branch, the address of the branch instruction and its target are stored as pairs. The number of pairs stored in the LBR queue varies among the x86 processor families. However, all families of processors since Pentium 4 record information about a minimum of four last branches taken. These values can be read from the MSR registers `MSR_LASTBRANCH_k_FROM_IP` and the `MSR_LASTBRANCH_k_TO_IP` where k is a number from 0 to 3. We check the branch that transferred execution to the entry gate using the LBR information. In the invocation checking routine, the second most recent branch is the one that was used to invoke the entry gate. We check that the source of the branch corresponds to the hook that is supposed to call the entry gate. Although the target of the branch instruction is also available, we do not need to verify it if the source matches. A conceivable attack may be an attempt to modify these MSR registers in order to bypass the invocation checks. We need to stop malicious modifications to these MSR, but at the same time ensure that performance requirement is not violated. With Intel VT, read and write accesses to MSR registers can selectively cause VMExits by setting the MSR read bitmap and MSR write bitmap, respectively. Using this feature, we set the bitmasks in such a way that write attempts to the `IA32_DEBUGCTL MSR` and the LBR MSRs are intercepted by the hypervisor but read attempts are not. Since the invocation checking routine only needs to read the MSRs, performance is not affected.

3.3.3 Trusted Path Execution

The core of the system is represented by the System Call Tracer component. Its goal is to intercept the operations performed by the OS, in terms of system call type, parameters and return values. All these information will be used by the Policy Matcher, to verify the right permissions on a certain resources on behalf of the process. There are two main requirements for this component: (S4) The interception mechanism must provide a trusted path between the invocation of the system call and its own termination. In particular the system needs to provide a secure hooking mechanism for intercepting the invocation and termination of OS operation. (P1) The overhead of the interception mechanism must be kept as low as possible. The trusted path is crucial for our work and it represents one of the main contributions of the extended version of the paper.

In order to retrieve all system call information, we need to monitor the invocation of the operation along with its own termination. Whenever a system call is issued by a process, a `sysenter` instruction is invoked. The `sysenter` instruction refers to the SYSENTER_EIP MSR that contains the address of the system call handler. In order to bring the execution flow inside the hypervisor, we need to switch from VMX non-root mode to VMX root mode. For this reason, we overwrite the SYSENTER_EIP MSR so that it points to a `vmcall` instruction. By using this hooking technique the hypervisor is able to intercept all the system calls performed by the OS and to parse the parameters according to their type. Note that any change of the MSR value on behalf of the system is intercepted and denied by the hypervisor. In this way, the system is able to protect the system call interception mechanism (requirement S4).

Before passing the information to the Policy Matcher, the system also needs to check whether the operation is successful or not and to collect its return value. For this purpose, our hypervisor is able to intercept a `sysexit` instruction by substituting it with a `vmcall`. Any attempt to re-write the VMX instruction is prevented by the hypervisor through a memory page protection mechanism (requirement S4). To verify the trusted path of the system call, our hypervisor also implements a simple automaton that checks the correctness of the system call execution flow. Every time a `sysenter` (entry gate) is intercepted an opening bracket "(" transition is triggered to indicate which system call was invoked. Every time a `sysexit` (exit gate) is intercepted, the hypervisor verifies that the watchpoint was expected, given the invoked system call. It performs this step repeatedly until it sees the watchpoint ")", corresponding to the end of the system call request. Any unknown state is reported as a system anomaly. If the operation succeeded, the System Call Tracer invokes the Policy Matcher component and provides all the information on the system call type, parameters, and return value.

Since the hypervisor is intercepting a high number of system calls, the hooking mechanism is a critical component from a performance point of view. Consequently, to improve performances, we devise two modifications to our original implementation. First, the system allocates a protected memory page that contains a short control code and some data about the monitored system calls—such as the system call types and the memory handler code address. Based on the system call type, the code decides whether to invoke a hypercall to switch to monitor mode or to leave the control-flow to the default system call handler. By using this technique we are able to exclude the non-monitored system calls and reduce the overhead of the whole hypervisor system (requirement P1). More details about performance evaluation are reported in the Section 6.

Another relevant source of overhead is related to possible multiple repetitions of the same system call from the same process. For example, during a file copy operation, the same read and write operations are repeated multiple times, according to the size of disk blocks and of copied file. Since there is no reason to check the permissions for each operation, our system is designed to verify only the first occurrence of the operation and run the other operations natively. The overhead caused by the repetition of these operations is thus avoided. This is implemented by introducing a small cache that contains a checksum based on the system call number, its parameters, and the value of the CR3 register of the process which is performing the operation. Every time the system discovers a new operation, we insert it into the cache and when the operation is not likely to be repeated (e.g., the corresponding process terminates, or the file is closed), we flush the cache entry related to that operation. In this way we only check the first operation and we skip possible repetitions (requirement P1). We report a measurement of the effectiveness of our cache in Section 6, Figure 6 and Figure 7.

3.3.4 Process Revealer

The goal of this component is twofold: First, it extracts and provides the name of the process that is performing the actual operation (i.e., a system call) through Virtual Machine Introspection [T. and M, 2003] and, second, it caches this information to reduce the system overhead. The component keeps a cache that allows to lookup the name of the process given a certain CR3 value. The cache is updated every time a process is created or destroyed, by properly intercepting and analyzing process-related system calls.

3.3.5 Policy Checker

The goal of this component is to check AccessMiner policies and to generate an alert in case some of them are violated. The policies are created by the model described in Section 2 and enforced system follows the model described in Section 2.5.

We recognize two main phases for the Policies Checker task: Initialization and Detection phase. The initialization phase is responsible to create the memory structures that will be used for the detection phase. In particular, to check the file system and registry policies, we adopt a hash table memory structure where the name of each resource is used as key and the name of process with its own permissions on that resource is stored as value. During the initialization phase, the hypervisor receives the signatures using the ad-hoc network communication protocol we briefly mentioned above. Then, whenever a signature is loaded, the full pathname of the corresponding resource is extracted and inserted in the memory structure as a key of the hash table. The list of the processes that can get access to the resource along with their own access permission are inserted as elements of such a key.

Another important memory structures used by the policy matcher is the *file/registry handles* structure. Since most of the file system and registry system calls operate on handles, while our policy system works with full-pathname resources, the system needs to keep the association between a handle number and the resource full pathname. For this reason, we use a dynamic memory structure that tracks this association. During the monitoring of the system, every time a resource is created or opened, the system retrieves the handle associated to the resource full pathname and it registers it in the structure. Afterwards, when a system call operates on the same handle, the corresponding object is retrieved from the handle structure. Every time a handle is closed, the system removes it from the handles memory structure.

To protect the policy information loaded during the initialization phase, the network driver that receives commands is only enabled when the hypervisor is in Management mode—in our prototype, this is triggered by using a special keystroke sequence. On the other hand, to protect the policies from network attacks, a signature scheme between the hypervisor and the management console is provided. In this way, we can assure that no one is able to tamper the hypervisor configuration information, according to our thread model.

During the detection phase, the System Call Tracer invokes the Policy Checker with the relevant system call information. At this point, the Policy Checker, by using the resources full-pathname as a key of the hash table, retrieves the list of the processes along their permissions. It also queries the process Revealer component in order to retrieve the processes name that acts as a subject of the operation. Once all the information is obtained, it scans the list of the processes to search the process name. If the process is not allowed to perform the operation, the Policy Checker raises an alert and blocks the operation. Otherwise, it permits the operation and then returns to non-root mode.

4 System Call Data Collection

In this section, we discuss our efforts to collect a large and diverse set of system call traces. Our requirements are geared towards imposing the least impact on the users whose machines are part of the data collection effort. Thus, the data collection framework must have minimal impact on the performance of those machines, must operate with and without network connectivity, must ensure that private information does not leave the user's machines, and must make almost no assumptions about the run-time environment. For example, requiring that users make use of virtual machines would significantly restrict the practical applicability of our data collection. Additionally, the data collection framework must be capable of extracting a rich set of attributes for each event (i.e., system call) of interest. Unfortunately, none of the existing system call tracing tools satisfies these requirements, so we built and deployed our own data collection framework.

Our system consists of a number of software agents, which, once installed on user's machines, automatically collect, anonymize, and upload system call logs, and a central data repository, which receives logs from each machine and normalizes the data in preparation for further analysis. The software agents can be installed by users on their own machines and are mindful of system load, available disk space, and network connectivity. Furthermore, users can enable and disable the collection agent as they wish.

Our analysis and training algorithms need several information regarding each system call. Therefore, our sensors were designed to collect the system call number and its arguments, its result (return) code, the process ID, the process name, and the parent process ID. Each log entry is represented by a tuple in the form:

$$\langle timestamp, program, pid, ppid, system\ call, args, result\rangle$$

This data allows us to perform our analyses within a single process, across multiple executions of the same program, or across multiple programs.

4.1 Raw Data Collection

The software agent that collects data is a real-time component running on each user's machine. This agent consists of a data collector and a data anonymizer. We implemented our agent for Microsoft Windows, as it is the OS targeted the most by malware. The description in the remainder of this section provides details specific to the Microsoft Windows platform. The data collector is a Microsoft Windows kernel

module that traces system call events and annotates them with additional process information. The data anonymizer transforms the collected system call data according to privacy rules and uploads it to the remote, central data repository. More in details the privacy rules used for our system are described in the section below called Log anonymizer.

Kernel collector. The main goal of this component is to collect system call and process information *across the entire system*. In order to intercept and log system call information, the kernel data collector hooks the SSDT table [Hoglund and Butler, 2005]. The kernel collector logs information for 79 different system calls in five categories: 25 related to files, 23 related to registries, 25 to processes and threads, one related to networking, and five related to memory sections. We selected the same subset of system calls that are used in Anubis [Anubis,], which covers the relevant operations that manipulate persistent OS resources.

A challenge arises from the fact that the kernel collector does not necessarily observe the start of a new process. One reason is that the user can disable and re-enable the software agent at any point. Another reason is that the kernel collector is started as the last kernel module in the system boot process. This means that the kernel collector might observe system calls that refer to previously acquired resource handles, but without having any information about which resources those handles point to. As a special case, some resource handles (e.g., handles to the registry roots) are automatically provided to a process by the OS at process-creation time. Consequently, if we log only the parameters for each individual system call that we observe, we lose information about previously (or automatically) acquired resources. To address this problem, we query the open handler table for each process we have not seen before. This allows the kernel collector to retrieve the open objects already associated with a new process. We store the path names of these objects for later use, for example when we intercept a system call (such as NtOpenKey) that references a pre-existing handle.

Log anonymizer. To protect the privacy of our users, we obfuscate or simply remove arguments of various system calls before sending the log to the data repository. The obfuscation consists of replacing part or the entire sensitive argument value with a randomly-generated value. Every time a value repeats, it is replaced with the same randomly-generated value, so that we can recover correlations between system call arguments. We consider as sensitive all arguments whose values specify non-system paths (e.g., paths under C:\Documents and Settings are sensitive), all registry keys below the user-root registry key (HKLM), and all IP addresses. Furthermore, we remove all buffers read, written, sent, or received, thus both providing privacy protection and reducing the communication to the data repository. The data repository indexes the logs by the primary MAC address of each machine.

Impact on performance. We designed the software agent to minimize the overhead on users' activities. The kernel module collects information only for a small subset of the 79 system calls. Log are saved locally and processed out of band before being sent to the server, when network connectivity is available. Users can turn data collection on and off, based on their needs. Local logs are uploaded to the repository when they reach 10 MB in size and logging is automatically stopped if available disk space drops below the 100 MB threshold. Each 10 MB portion of the system call log is compressed using ZIP compression, for a 95% average reduction in size (from 10 MB to 500 KB). Given these techniques, we are confident that users were able to use their computers with the data collector present as they would normally do, and thus the collected system call logs are representative of day-to-day usage.

4.2 Data Normalization

The purpose of this component is to process the raw system call logs and extract the fully qualified names of the accessed resources as well as the access type. For files and directories, the fully qualified name is the absolute path, while for registry keys it is the full path from one of the root keys.

To compute fully qualified resource names, we track for each process the set of resources open at any given time, via the corresponding set of OS handles. When a resource (file or registry key) is accessed relative to another resource (either opened by the process or opened by the OS automatically for the process), we combine the resource names to obtain a fully qualified name.

Computing the access type (e.g., read, write, or execute) requires tracking the access operations performed on a resource. This is more tricky than expected. When a resource is acquired by a program (e.g., a file is opened), the program specifies a desired level of access. This information, however, is not sufficiently precise for our needs. This is because, often, programs open files and registry keys at an access level beyond their needs. For example, a program might open a file with FULL_ACCESS (i.e., both read and write access), but afterward, it only reads from the file. Since we are interested in the actual access type, we track all of the operations on a resource, and only when the resource is released (on NtClose), we compute the access type as a union of all operations on the resource.

In Microsoft Windows, there is no single system call that starts a new process from a given executable file. In order to retrieve the execution path and file name, the normalizer needs to recognize the NtOpenFile system calls that belong to the process-creation task. When a process is created, the OS executes a set of system calls to allocate resources, load the binary executable, and start the new process: NtOpenFile, NtCreateSection with desired executable access, and NtCreateThread. Consequently, we automatically identify occurrences of this pattern and extract the executable path and file name.

4.3 Experimental Data Set

We used different datasets in our experiments. The first is a collection of execution traces of 6,000 malware samples randomly extracted from Anubis [Anubis,]. This set, that we call malware, includes a mix of all the existing categories (e.g., botnets, worms, dropper, Trojan horses), extrapolated from

Machine	*Data* (GB)	*System calls* $(\times 10^6)$	*Processes* $(\times 10^3)$	*Applications*
1	18.0	285	55.1	90
2	4.5	70	22.4	87
3	5.6	89	17.7	46
4	32.0	491	110.9	41
5	34.0	514	125.6	42
6	14.0	7	2.8	73
7	1.3	19	3.7	49
8	1.2	18	3.0	22
9	1.6	27	8.5	47
10	2.3	36	12.9	26
Total	114.5	1,556	362.6	242

TABLE 1
Characteristics of our Data Set.

Machine	*Usage*	*Data*	*Time*		*Data rate*
			Logged	*Total*	
		(GB)	(hours)	(days)	(MB/minute)
1	office	18.0	12	3	8
2	home	4.5	4	3	6.25
3	home	5.6	3	4	7.77
4	prod.	32.0	12	3	14
5	prod.	34.0	12	3	15
6	lab	14.0	8	3	11
7	home	1.3	3	2	4
8	home	1.2	3	2	4
9	dev.	1.6	2	2	6
10	dev.	2.3	2	3	6.4

TABLE 2
Data Rates During Collection.

malware that is active in the wild. The second dataset contains 114.5 GB of execution traces collected from 10 different real-world machines, where we observed normal day-to-day operation of end-users computer. In particular, the benign data consists of 1.556 billion of system calls, from 362,600 processes and 242 distinct applications. In table 1 we provide detailed information for each machine. The choice of the nine machines used for model construction is done using 10-fold cross-validation approach. The evaluation results presented here are averages across the 10 tests. We deployed our data collection framework on ten different Windows machines, each belonging to a different user. The users had different levels of computing expertise and different computer usage patterns. Based on their role, the machines can be classified as follows: two development systems, one office system, one production system, four home PCs, and a computer-lab machine.

Our system collected data from each machine at an average rate of 8.2 MB/minute, with highly used machines producing logs at 40 MB/minute and idle machines producing 1.5 MB/minute. In table 2, we report the logging time for the ten different machines. For each machine, we show the machine's usage profile, the size of data collected, the total time during which data was actually collected, the time period between the first log entry and the last log entry, and the average data rate. For example, the fourth row indicates that machine 4 was a production server that generated 32 GB of system call logs, over a period of 3 days, during which data collection was active for 12 hours.

5 Detection Results

In this section, we evaluate the effectiveness of our system in detecting malicious activities on real systems.

More precisely, we conducted ten experiments. For each one, we selected one of the machines and we used the system calls recorded on the other nine hosts to generate the access activity model, as described in Section 2. Finally, we used this model for detection by checking the resource accesses performed by all processes on the machine that was *not* used for model generation. Then, we examine the accesses performed by the malware samples. For each experiment, we evaluate the detection capabilities and false positives of the file system model alone, the registry model alone, and both models combined.

5.1 File System Access Activity Model

On average, the file system access activity model contains about 100 labels. These labels contain tokens that restrict read access to about 70 directories, write access to about 80 directories, and execute access to about 30 directories. The results for the file system model are shown in Table 3. In this table, we see a number of different columns for the detection rates and the false positive rates. These are discussed in the following paragraphs.

When using the original model to check all read, write, and execute accesses, we see an average detection rate of 66% for the malware samples (column *Detection rate*) and a false positive rate of almost 15% (column *False positive rate*). Note that the false positive rates are computed on the basis of single applications and not on a process basis.

At first glance, the results appear sobering. However, a closer examination of the result reveals interesting insights. First, we decided to investigate the false negative rate in more detail. When looking at the execution traces of the malware programs, we observed that many samples did not get far in their execution but quickly exited or crashed. Interestingly, a substantial fraction of suspicious samples never wrote to the file system or the registry, and they did not open any network connections. It is difficult to confirm that these samples exhibit any malicious activity at all. As a result, we decided to remove from our malware data sets all samples that never perform a write operation or open a network connection. This decreases our malware data set to 7,847 samples that exhibit at least some kind of activity. It also improves our detection rate to more than 90%, as reported in column *Adjusted detection rate* of Table 3. For the remainder of this paper, all reported detection rates are computed based on the adjusted malware data set.

In the next step, we investigated the false positives in more detail. Table 3 shows the access violations for each machine, divided into violations due to read (column *Read*), write (column *Write*), and execute (*Execute*) access attempts. It can be seen that execute violations account for a significant majority of false positives. However, we also found that they are only marginally important for detection. Thus, for the next experiment, we decided to use only the access tokens that refer to write operations. This is justified by the fact that we

	Experiment 1		Experiment 2				
Machine	*Detection rate*	*False positive rate*	*Adjusted detection rate*	*Rates of detected access violations*			*Detection rate (only writes)*
				Read	*Write*	*Execute*	
1	0.656	0.225	0.906	0.000	0.022	0.222	0.864
3	0.657	0.154	0.907	0.000	0.130	0.043	0.902
4	0.657	0.156	0.907	0.024	0.049	0.122	0.902
5	0.657	0.143	0.907	0.024	0.024	0.095	0.902
6	0.635	0.242	0.877	0.014	0.055	0.242	0.868
7	0.657	0.267	0.907	0.020	0.041	0.265	0.901
8	0.657	0.045	0.907	0.000	0.045	0.000	0.902
9	0.657	0.025	0.907	0.000	0.025	0.000	0.902
10	0.657	0.050	0.907	0.000	0.038	0.038	0.902
Average	0.655	0.148	0.904	0.008	0.044	0.137	0.895

TABLE 3
Partial Detection Based on our Filesystem Access Activity Model.

Machine	*FP rate*	*Final det.rate*
1	0.0	0.864
2	0.0	0.902
3	0.0	0.902
4	0.0	0.902
5	0.0	0.902
6	0.0	0.868
7	0.0	0.901
8	0.0	0.902
9	0.0	0.902
10	0.0	0.902
Average	0.0	0.895

TABLE 4
Final Detection Based on our Filesystem Access Activity Model.

are most interested in preserving the integrity of the operating system resources. The detection results for the new *write-only* detection approach are presented in column *Detection rate (only writes)* of Table 3. As can be seen, the numbers remain high with 89.5%. This confirms that write access violations are a good indicator for malicious activity. With this approach, the false positives are identical to the write violations, which are shown in column *Write*.

We further examined the reasons for the remaining write violations. It turned out that these violations were due to two root causes. One set of false positives was caused by our own system-call logging component that wrote temporary files directly into the C:\ directory before sending the data over the network. The second violation was due to software updates. More precisely, we detected a number of cases in which an application was writing to its folder in C:\Program Files. Of course, only this program had read/execute access to that directory. However, we never saw a write access during training, and as a result, the directory was considered read-only. To accommodate for updates, we manually added a rule to the model that would grant write permission to applications that "own" directories in C:\Program Files. Moreover, we granted our component write access to C:. With more extensive training, both access activities would have very likely been added automatically. The model that incorporated our minor adjustments generated no more false positives, as shown in Table 4. However, the detection capabilities of the model remain basically unchanged, as shown in Table 4.

Machine	*Detection rate*	*False positive rate*	*Det. rate (only writes)*	*FP rate (only writes)*	*Final det. rate*
1	0.567	0.063	0.530	0.063	0.521
2	0.557	0.107	0.540	0.053	0.521
3	0.566	0.179	0.530	0.128	0.062
4	0.557	0.000	0.530	0.000	0.540
5	0.557	0.000	0.530	0.000	0.540
6	0.557	0.015	0.530	0.000	0.540
7	0.597	0.133	0.530	0.000	0.540
8	0.557	0.067	0.530	0.067	0.537
9	0.561	0.100	0.530	0.025	0.521
10	0.557	0.000	0.530	0.000	0.540
Average	0.563	0.066	0.530	0.034	0.486

TABLE 5
Detection Based on our Registry Access Activity Model.

5.2 Registry Access Activity Model

In our experiments, the registry access activity model contained in average about 3,000 labels, significantly more than the file-system model. In particular, the labels contained tokens that restrict read access to about 1,600 keys and write access to about 2,800 keys (*execute* is not defined for registry keys).

The results for the registry model are shown in Table 5. The columns *Detection rate* and *False positive rate* show the detection rates and the false positive rate, respectively, for the

original model. It can be seen that both the detection rate and the false positive rates are lower than for the file system model. We also examined the detection rate and the false positive rate when considering only write operations (columns *Det. rate (only writes)* and *FP rate (only writes)*). Similar to the file system case, the false positive rate drops significantly; there are five runs in which no false positives were reported at all. However, the detection rate remains (relatively) high.

We also examined the cases for which the registry access model raises false positives. We found that all registry write access violations can be attributed to the sub-tree HKEY_USERS\Software\ Microsoft. While this is an important part of the registry that contains a number of security settings, we wanted to understand the detection capabilities of a model that permits write access to these keys. To this end, we added a manual rule to allow writes to this sub-tree and re-run the experiments on the malware data set. We see that the model is still effective and achieves a detection rate of over 48% (shown in column *Final det. rate* of Table 5) with no false positives. Considering the significantly larger size of the registry models compared to the ones for the file system, we expect that a larger training set would be required to effectively capture legitimate writes to the Software\Microsoft sub-tree.

5.3 Full Access Activity Model

For the final experiment, we combined those improved file system and registry models that yielded zero false positives. The combined detection rate improves compared to the file system model alone, but only slightly (between 1% and 2% for all of the ten runs). The average detection improved from 89.5% to 91% (of course, with no false positives).

5.4 Discussion.

When focusing on write operations only, our access activity model achieves a good detection rate (more than 90%) with a very low false positive rate. The false positive rate even drops to zero with minor manual adjustments that compensate for deficiencies in the training data, while still retaining its detection capabilities. This suggests that a system-centric approach is suitable for distinguishing between benign and malicious activity, and it handles well even applications not seen previously. This is because most benign applications are written to be good operating system "citizens" that access and manage resources (files and registry entries) in the way that they are supposed to.

Malicious programs frequently violate good behavior, often because their goals inevitably necessitate tampering with system binaries, application programs, and registry settings. Of course, we cannot expect to detect all possible types of malicious activity. In particular, our detection approach will fail to identify malware programs that ignore other applications and the OS (e.g., the malware does not attempt to hide its presence or to gain control of the OS) and that carry out malicious operations only over the network. Most of the 10% of malware that represent the false negative was waiting for a particular command from the network in order to perform malicious actions (botnet, spammer etc.), or they were waiting

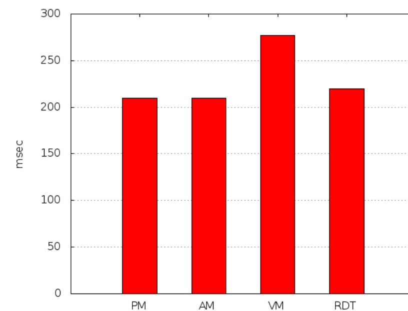

Fig. 3. Memory Read Operation.

for a particular conditions in order to activate themselves. In general this malware are designed to hide information to the user. It is important to note that the malicious system call traces were extracted from Anubis, where the system only wait for a small time window for analyzing a single sample (20 minutes). Consequently some malware did not have a chance to perform their own malicious actions and cannot be detected by AccessMiner.

6 Performance Results

In this section we report a set of micro and macro benchmarks we used to demonstrate the efficiency of our new system. In our experiments we run the Passmark Performance Test suite [Software,] in four different test environments: on a physical machine (PM), inside a guest VMWare virtual machine (VM), on physical machine with AccessMiner (AM), and on physical machine running the Hypersight (RTD) [Northsecuritylabs,] real-time rootkit detector. Hypersight is a hardware-supported virtual machine monitor that starts at boot time and intercepts several types of suspicious actions applied to critical memory structures such as attempts to modify page tables, read-only kernel modules, and GDT and IDT tables. All the experiments are performed on an Intel Core i7 2.67 GHz with 3 GB of memory running a Windows XP (32-bit) OS.

6.1 Macro Benchmark

We measured and compared the overhead introduced by AccessMiner on different workloads by using four of the PassMark performance tests: memory operations read and write, and sequential disk read and write operations. To perform these tests, we loaded AccessMiner with 3824 policy rules: 173 signatures related to the file system and 3651 signatures related to the Windows registry.

The final values were obtained by taking the average of ten repetitions for each benchmark. Figure 3 and Figure 4 show the results of the memory tests. In these cases, we used the system to perform a sequential read or write operation of 1GB of memory with a block size ranging from 1024 bytes up to 512 MB. As we can see in the graph, the higher overhead is encountered in VmWare, mainly due to its memory virtualization. AccessMiner does not introduce any overhead, since it does not virtualize the memory but only uses a memory

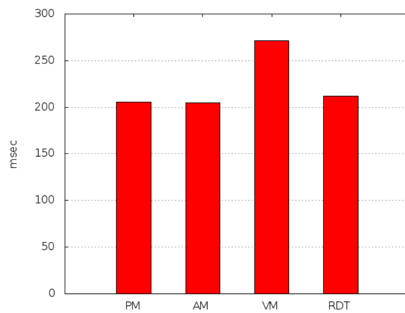

Fig. 4. Memory Write Operation.

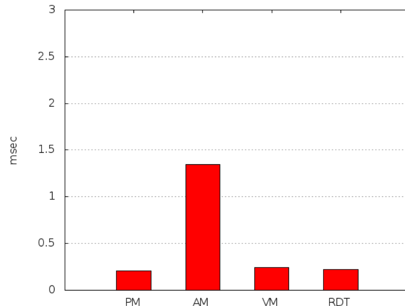

Fig. 5. Disk Read Operation.

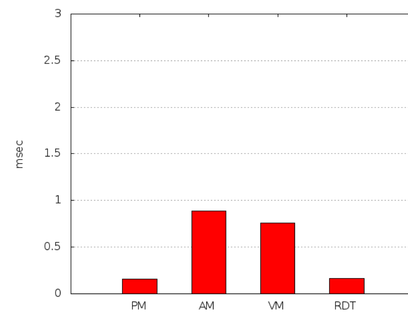

Fig. 6. Disk Write Operation.

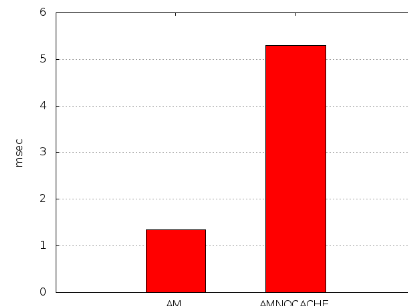

Fig. 7. Cache/NoCache Disk Read Operation.

protection mechanism. The overhead introduced by Hypersight is due to the memory scanning operation responsible to check the sensible memory structures.

More interesting performance results are reported in Figure 5 and Figure 6. For this test, we performed a sequential read and write *disk* operations for 1 minute. We use the NTFS file system with a block size of 8192 bytes. As we can see in the graphs, the overhead of our system is similar to the one observed in a VmWare virtual machine, while the Hypersight overhead is the same of the physical machine, since it does not intercept any operation on the disk. The overhead for our system is due to the high number of file system and registry system calls performed by the benchmark program. However, in these scenarios with multiple repetitive operations our caching mechanism is able to reduce the overhead of almost 80%, as reported in Figure 7 and Figure 8.

During the PassMark disk test we counted 11.000 `NtRead/NtWrite` system calls related to the file system operations and other 5430 system calls related to the registry operations. The impact of the disk operation was largely covered by our caching system, leaving the registry responsible for most of the overhead.

In table 6 we report the overhead of memory operations for the four environments test: PM, AM, VM, and RTD system. In table 7 we report the overhead of the disk operations.

To conclude the macro benchmarks, we also performed a worse-case experiment, in which we measured the overhead introduced by AccessMiner during a source code compilation routine. The target of the compilation was a middle-sized C program, composed of almost 100,000 lines of code. Results of this last benchmark are reported in Figure 9. In this case, since the IO operations were spread on hundreds of different files, our caching mechanism was less effective in mitigating the disk overhead. This resulted in an average AccessMiner overhead, with respect to the physical machine baseline, of around 2.5x. It is important to note that our system can be tuned to obtain better performance. For instance, a possible optimization could be to monitor only untrusted applications that are downloaded from untrusted sources such as network or copied from untrusted devices. Monitor only a small set of applications can improve the performance without losing the detection rate. It is important to note that in our design, it is enough to remove an application from the monitored set to exclude it from further analysis.

To conclude, the performance of AccessMiner greatly depend on the type of application. However, the system normally introduces an overhead, considering memory and disk operations together, that is comparable with the one observed in a traditional virtual machine environment.

6.2 Micro-Benchmarks

To have a more fine-grained view of the delay introduced by our system, we measured the overhead introduced by triggering a system call on a particular resource. We started by measuring the time needed to perform a context switch between a VM exit and a VM entry (without checking any policy), taking an average over 20 repetitions. The operation took 1216 clock cycles, corresponding to around 0.45 microseconds. The second operation that we measured was the entire syscall monitoring mechanism. In this case, the time needed to intercept a single system call is, in average, 1,241,739 clock cycles, or about 0.47ms. These results show that most of the overhead introduced by our new system is

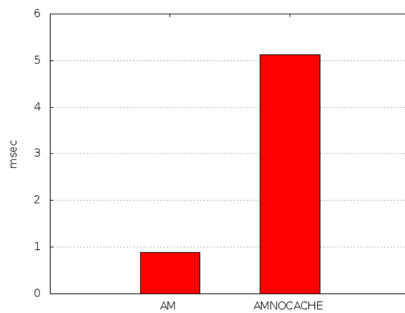

Fig. 8. Cache/NoCache Disk Write Operation.

Memory Operations	PM	AM	VM	RTD
Read (ms)	209.8916	209.9269	277.2247	220.0006
Write (ms)	205.5485	205.1042	271.5995	212.1548
Overhead Read	NA	100.0 %	132.1 %	104.8 %
Overhead Write	NA	99.9 %	132.1 %	103.2 %

TABLE 6
Overhead of Memory Operations.

due to the policy validation mechanism, while the context switch along with the monitoring mechanism does not impact the system in a relevant way. Such results highlight the high efficiency of our new system that is built on top of the AccessMiner model.

7 Related Work

The existing papers most relevant to our current work focus on malware detection at the system call and the system library interfaces. These interfaces best describe the system resources manipulated by a program (e.g., files, other programs, other processes, configuration data, authentication and authorization information, network communication channels), making system call-based detectors comparable to our access activity model.

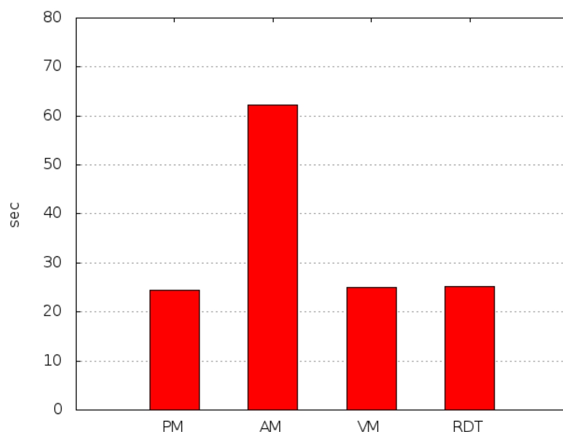

Fig. 9. GCC Evaluation.

Disk Operations	PM	AM	VM	RTD
Read (ms)	0.2070	1.3470	0.2460	0.2240
Write (ms)	0.1580	0.8900	0.7590	0.1640
Overhead Read	NA	650.7%	118.8%	108.2%
Overhead Write	NA	563.3 %	480.4 %	103.80 %

TABLE 7
Overhead of Disk Operations.

7.1 Malware Detection

Malware detection has looked at many ways to describe program behavior, and corresponding models evolved to keep pace with the increasing complexity of malware. Early detection mechanisms were based on particular byte sequences in the program binary that were indicative of malware. Over time, obfuscation strategies pursued by malware writers forced detectors to move to regular expressions over bytes [Ször, 2005], and eventually rendered them obsolete as byte patterns have little predictive power (i.e., they can accurately capture only previously seen malware). Other models such as byte n-grams [Li et al., 2005], system dependencies of the program binary [Schultz et al., 2001], and syntactic sequences of library calls [Xu et al., 2004], [Mukkamala et al., 2004] have been proposed with limited success. Because these models have a strong syntactic aspect in that they capture artifacts of program binary unrelated to the malicious behavior, malware writers managed to evade such defenses and produce new, undetected malware. Our emphasis on a system-centric approach to modeling resource interactions bypasses such syntactic artifacts.

The software-diversity tactics employed by malware writers required new detection techniques that could capture more of the intent of the program and less of the syntactic characteristics of the program binary. The research efforts have focused on describing malware in terms of violations to an information-flow policy. Because it is not feasible for performance reasons to track system-wide information flows accurately, the focus shifted on better and better approximations of the information flow. Bruschi *et al.* [Bruschi et al., 2006] and Kruegel *et al.* [Kruegel et al., 2005b] showed that some classes of obfuscations could be rendered innocuous by modeling programs according to their instruction-level control flow, while Christodorescu *et al.* [Christodorescu et al., 2005] and Kinder *et al.* [Kinder et al., 2005] built obfuscation-resilient detectors based on instruction-level information flow. Nonetheless, instruction sequences are fungible and there are many ways to implement the same high-level functionality. Detection techniques then raised the bar by capturing information flow at the level of library calls, as proposed by Kirda *et al.* [Kirda et al., 2006], system calls, as proposed by Kolbitsch *et al.* [Kolbitsch et al., 2009], Christodorescu *et al.* [Christodorescu et al., 2007], Martignoni *et al.* [Martignoni et al., 2008], and Stinson *et al.* [Stinson and Mitchell, 2007], and OS resources, as proposed by Yin *et al.* [Yin et al., 2007]. The respective evaluations of each of these techniques shows that as the models used in detection more closely describe actual OS resources, the detection rates significantly increase and the false-positive rates decrease. Unfortunately the library

and system-call interfaces are rich enough that mimicry attacks are still possible [Kruegel et al., 2005a], [Wagner and Soto, 2002]. This observation guided our choice of system resources as the basic element in our models, discarding any information about the order in which resources are accessed. Furthermore we focus strictly on system resources that are shared across processes (i.e., files, registry, and network connections) and we ignore single-process resources such as virtual memory.

Beyond proposing a richer, system-centric model of program behavior, we made a concerted effort to improve an often overlook evaluation aspect, the external validity of the experimental settings. This concerns the number and diversity of benign and malicious programs used to evaluate a detection technique, as well as the environment in which they are exercised (in the case of detectors that rely on runtime information). For example, Kirda *et al.* [Kirda et al., 2006] evaluated their system against 33 malware samples and 18 benign samples, each samples executed for 30–60 seconds. Kolbitsch *et al.* [Kolbitsch et al., 2009] used 563 malware samples and 10 benign samples, executed for up to 5 minutes. Christodorescu *et al.* [Christodorescu et al., 2007] evaluated 16 malware samples and 6 benign samples for up to 4 minutes, similar to the test sets used by Martignoni *et al.* [Martignoni et al., 2008] (7 malware, 6 benign) and to Stinson *et al.* [Stinson and Mitchell, 2007] (6 malware, 9 benign). Yin *et al.*, in their PANORAMA system, evaluated 42 malicious samples and 56 benign ones, for 5 minutes. What is common to all of these evaluations is that both the numbers of malicious samples and of benign samples are quite small. On current systems, regular users often run tens of interactive applications and hundreds of background processes, casting doubt on the relevance of results obtained from a few benign samples. Furthermore, evaluations in previous work were performed in virtualized, constrained environments, where interactive applications were exercised mechanically in ways that do not necessarily reflect real-life usage. We addressed these limitations by collecting execution traces of benign applications from actual users, during the course of their normal interaction with their personal systems. We designed our system to have low overhead and to anonymize all collected information, so that the users had no concerns and were not impacted in their regular use of their machine. The benign data we collected covered 242 distinct benign applications ran by ten users in their own environments.

7.2 Malware Classification

Another research topic that is closely related to our work is that of classification of large sets of malware samples. Various models have been proposed, all focusing on system calls or on accesses to system resources. Lee and Mody performed classification of malware samples based on the similarity between sequences of system calls [Lee and Mody, 2006]. Bailey *et al.* [Bailey et al., 2007] considered similarity between sets of accessed system resources, and Rieck *et al.* [Rieck et al., 2008] considered various refinements by abstraction. Bayer *et al.* [Bayer et al., 2009] used similarity between resource-based information flows for classification. All of these papers describe the classification task applied to large sets of malware (thousands or tens of thousands), and thus their results are representative. Yet, because their primary focus was on malware classification, it is not clear that the classification features that they derived are useful in malware detection. A classification feature (e.g., some particular resource accesses) might well distinguish botnet M_1 from botnet M_2, but it might not be able to distinguish botnet M_1 from a benign program B. Thus our current work is orthogonal to malware-classification research.

7.3 Access Control and Domain and Type Enforcement

Our system-centric access activity model is related to access control mechanisms, and, in particular, to mandatory access control (MAC) systems. They both define acceptable uses of resources in a user-independent way via a central policy. There are numerous implementations of MAC systems, of which SELinux [Loscocco and Smalley, 2001] is currently the most visible. Some MAC systems have been specifically designed to prevent malware from running in a system [Salois and Charpentier, 2000], [Debbabi et al., 2001], while others can enforce multi-level security policies. Based on this similarity, the system-centric model can be converted into a SELinux policy, for example, and our model-generation technique can be used as a practical tool to construct SELinux policies.

There is a fundamental distinction between MAC policies and our system-centric models. While a MAC policy necessarily enumerates all the programs and the program-specific rules, a system-centric model is more general in that it defines confidentiality and integrity rules for all programs. While it might appear that system-centric models are less restrictive, in our experimental evaluation, we observed a very good match between our models and real-life application executions. Additionally, MAC policy are often deployed to ensure the confidentiality and integrity of system files, at the cost of leaving user files poorly (if at all) secured and in need of additional mechanisms, such as the PinUP tool proposed by Enck *et al.* [Enck et al., 2008], which ties user files to particular applications. Our system-centric model covers system *and* user files, based on the observation that both system programs and applications satisfy some general ways in which they use OS resources.

7.4 Virtualization

The idea of utilizing a virtual machine monitor to perform sophisticated run-time analyses, with the guarantee that the results cannot be tampered by a malicious attacker, has already been widely explored in the literature. Garfinkel et al. were the first to propose to use a VMM to perform OS-aware introspection [T. and M, 2003]. Other researchers proposed to use a VMM for protecting the guest OS from attacks by monitoring its execution, with a software-based VMM [R. et al., 2008] that leveraged on hardware support for virtualization [A. et al., 2007]. Similar ideas were also proposed by other authors [B.D. et al., 2008], [M. et al., 2009]. In [X. et al., 2008] Chen et al. described a solution to protect applications' data even in the presence of a compromised operating system. Recently,

Vasudevan et al. proposed XTREC, a lightweight framework to record securely the execution control flow of code running in an untrusted system [A. et al., 2010]. Finally, our Hypervisor is lightweight version of HyperDbg [Fattori et al., 2010] and it can provide a secure layer for checking the AccessMiner Policies. It also provided a manage mode where the policies can be loaded in secure way from the network so we can assure that the attacker cannot modify them.

8 Conclusions

In this paper we present AccessMiner, a system-centric approach to model the activities of benign programs and use these models to detect the presence of malicious applications. In particular, we discuss the general algorithm and the implementation of the AccessMiner detector as a custom system hypervisor. We also discuss the accuracy of our approach and the overhead introduced by our hypervisor. The results of our experiments show that our system could be deployed in a real environment, with only a limited impact on the performance of the system.

References

[A. et al., 2010] A., P., V., G., and A., V. (2010). XTREC: secure real-time execution trace recording and analysis on commodity platforms. In *Technical report, Carnegie Mellon University (2010)*.

[A. et al., 2007] A., S., M., L., N., Q., and A., P. (2007). SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

[AMD, Inc.,] AMD, Inc. AMD Virtualization. www.amd.com/virtualization.

[Anubis,] Anubis. Anubis. http://anubis.iseclab.org.

[Bailey et al., 2007] Bailey, M., Oberheide, J., Andersen, J., Mao, Z. M., Jahanian, F., and Nazario, J. (2007). Automated classification and analysis of internet malware. In Kruegel, C., Lippmann, R., and Clark, A., editors, *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, volume 4637 of *Lecture Notes in Computer Science*, pages 178–197, Gold Goast, Australia. Springer-Verlag.

[Bayer et al., 2009] Bayer, U., Comparetti, P. M., Hlauschek, C., Kruegel, C., and Kirda, E. (2009). Scalable, behavior-based malware clustering. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS'09)*, San Diego, CA, USA.

[B.D. et al., 2008] B.D., P., M., C., M., S., and W., L. (2008). Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[Bruschi et al., 2006] Bruschi, D., Martignoni, L., and Monga, M. (2006). Detecting self-mutating malware using control-flow graph matching. In Büschkes, R. and Laskov, P., editors, *Proceedings of the 3rd Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'06)*, volume 4064 of *Lecture Notes in Computer Science*, pages 129–143. Springer-Verlag.

[Canali et al., 2012] Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. (2012). A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*.

[Christodorescu et al., 2005] Christodorescu, M., Jha, S., Seshia, S. A., Song, D., and Bryant, R. E. (2005). Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P'05)*, pages 32–46, Oakland, CA, USA. IEEE Computer Society.

[Christodorescu et al., 2007] Christodorescu, M., Kruegel, C., and Jha, S. (2007). Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*, pages 5–14, New York, NY, USA. ACM Press.

[Debbabi et al., 2001] Debbabi, M., Girard, M., Poulin, L., Salois, M., and Tawbi, N. (2001). Dynamic monitoring of malicious activity in software systems. In *Proceedings of the Symposium on Requirements Engineering for Information Security (SREIS'01)*, pages 1–10, Indianapolis, IN, USA.

[Enck et al., 2008] Enck, W., McDaniel, P. D., and Jaeger, T. (2008). Pinup: Pinning user files to known applications. In *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC'08)*, pages 55–64, Anaheim, CA, USA. IEEE Computer Society.

[Fattori et al., 2010] Fattori, A., Paleari, R., Martignoni, L., and Monga, M. (2010). Dynamic and transparent analysis of commodity production systems. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*. http://code.google.com/p/hyperdbg/.

[Hoglund and Butler, 2005] Hoglund, G. and Butler, J. (2005). *Rootkits: Subverting the Windows kernel*. Addison-Wesley Professional.

[Kinder et al., 2005] Kinder, J., Katzenbeisser, S., Schallhart, C., and Veith, H. (2005). Detecting malicious code by model checking. In Julisch, K. and Kruegel, C., editors, *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187, Vienna, Austria. Springer-Verlag.

[Kirda et al., 2006] Kirda, E., Kruegel, C., Banks, G., Vigna, G., and Kemmerer, R. (2006). Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium (Security'06)*, Vancouver, BC, Canada.

[Kolbitsch et al., 2009] Kolbitsch, C., Milani, P., Kruegel, C., Kirda, E., Zhou, X., and Wang, X. (2009). Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security'09)*, pages 351–366, Montréal, Canada. USENIX Association.

[Kruegel et al., 2005a] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. (2005a). Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium (Security'05)*, Baltimore, MD, USA.

[Kruegel et al., 2005b] Kruegel, C., Kirda, E., Mutz, D., Robertson, W., and Vigna, G. (2005b). Polymorphic Worm Detection Using Structural Information of Executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05)*, volume 3858 of *LNCS*, pages 207–226, Seattle, WA. Springer-Verlag.

[Lanzi et al., 2010] Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., and Kirda, E. (2010). AccessMiner: Using system-centric models for malware protection. In *Proceedings of the 17th Conference on Computer and Communications Security (CCS)*.

[Lee and Mody, 2006] Lee, T. and Mody, J. J. (2006). Behavioral classification. In *Proceedings of the 15th Annual European Institute for Computer Antivirus Research Conference (EICAR'06)*.

[Li et al., 2005] Li, W.-J., Wang, K., Stolfo, S. J., and Herzog, B. (2005). Fileprints: Identifying file types by n-gram analysis. In *Proceedings of the 6th Annual IEEE Systems, Man, and Cybernetics (SMC) Workshop on Information Assurance*, pages 64–71, West Point, NY. United States Military Academy.

[Litty et al., 2008] Litty, L., Lagar-Cavilla, H. A., and Lie, D. (2008). Hypervisor support for identifying covertly executing binaries. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 243–258, Berkeley, CA, USA. USENIX Association.

[Loscocco and Smalley, 2001] Loscocco, P. and Smalley, S. (2001). Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track of the 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA. USENIX Association.

[M. et al., 2009] M., S., W., L., W., C., and A., L. (2009). Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security*.

[Martignoni et al., 2010] Martignoni, L., Paleari, R., and Bruschi, D. (2010). Conqueror: tamper-proof code execution on legacy systems. In *Proceedings of the 7th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Lecture Notes in Computer Science, Bonn, Germany. Springer.

[Martignoni et al., 2008] Martignoni, L., Stinson, E., Fredrikson, M., Jha, S., and Mitchell, J. C. (2008). A layered architecture for detecting malicious behaviors. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection (RAID'08)*, pages 78–97, Berlin, Heidelberg. Springer-Verlag.

[McCune et al., 2008] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. (2008). Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*.

[Mukkamala et al., 2004] Mukkamala, S., Sung, A., Xu, D., and Chavez, P. (2004). Static analyzer for vicious executables (SAVE). In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 326–334, Tucson, AZ, USA.

[Neiger et al., 2006] Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. (2006). Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(3):167–177.

[Northsecuritylabs,] Northsecuritylabs. HyperSigh rootkit detector. http://northsecuritylabs.com/downloads/whitepaper-html/.

[R. et al., 2008] R., R., X., J., and D., X. (2008). Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection. (2008)*.

[Rieck et al., 2008] Rieck, K., Holz, T., Willems, C., Düssel1, P., and Laskov, P. (2008). Learning and classification of malware behavior. In Zamboni, D., editor, *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'08)*, volume 5137 of *Lecture Notes in Computer Science*, pages 108–125. Springer-Verlag.

[Salois and Charpentier, 2000] Salois, M. and Charpentier, R. (2000). Dynamic detection of malicious code in COTS software. In *Proceedings of the Information Systems Technology Panel (IST) Symposium on Commercial Off-the-Shelf Products in Defence Applications "The Ruthless Pursuit of COTS"*, pages 16–1—16–13, Brussels, Belgium. NATO Research and Technology Organization.

[Schultz et al., 2001] Schultz, M. G., Eskin, E., Zadok, E., and Stolfo, S. J. (2001). Data mining methods for detection of new malicious executables. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (S&P'01)*, pages 38–49.

[Software,] Software, P. PassMark Performance Test.

[Stinson and Mitchell, 2007] Stinson, E. and Mitchell, J. C. (2007). Characterizing bots' remote control behavior. In Kruegel, C., Lippmann, R., and Clark, A., editors, *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID'07)*, volume 4637 of *Lecture Notes in Computer Science*. Springer-Verlag.

[Symantec, 2011a] Symantec (2011a). W32.Duqu: The Precursor to the Next Stuxnet. http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet.

[Symantec, 2011b] Symantec (2011b). W32.Stuxnet Dossier. http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[Symantec, 2012] Symantec (2012). Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East. http://www.symantec.com/connect/blogs/flamer-highly-sophisticated-and-discreet-threat-targets-middle-east.

[Ször, 2005] Ször, P. (2005). *The Art of Computer Virus Research and Defense*. Addison-Wesley.

[T. and M, 2003] T., G. and M, R. (2003). Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Network and Distributed Systems Security Symposium, The Internet Society (2003)*.

[Wagner and Soto, 2002] Wagner, D. and Soto, P. (2002). Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM conference on Computer and Communications Security (CCS'02)*, pages 255–264, New York, NY, USA. ACM.

[Wojtczuk, 2008] Wojtczuk, R. (2008). Subverting the Xen hypervisor. *Black Hat USA*, 2008.

[X. et al., 2008] X., C., T., G., E.C., L., P., S., C.A., W., D., B., J., D., and D.R.K, P. (2008). Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *Operating Systems Review*.

[Xu et al., 2004] Xu, J., Sung, A. H., Chavez, P., and Mukkamala, S. (2004). Polymorphic malicious executable scanner by API sequence analysis. In *Proceedings of the 4th International Conference on Hybrid Intelligent Systems (HIS'04)*, pages 378–383, Kitakyushu, Japan. IEEE Computer Society.

[Yin et al., 2007] Yin, H., Song, D., Egele, M., Kruegel, C., and Kirda, E. (2007). Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS'07)*, pages 116–127, New York, NY, USA. ACM.